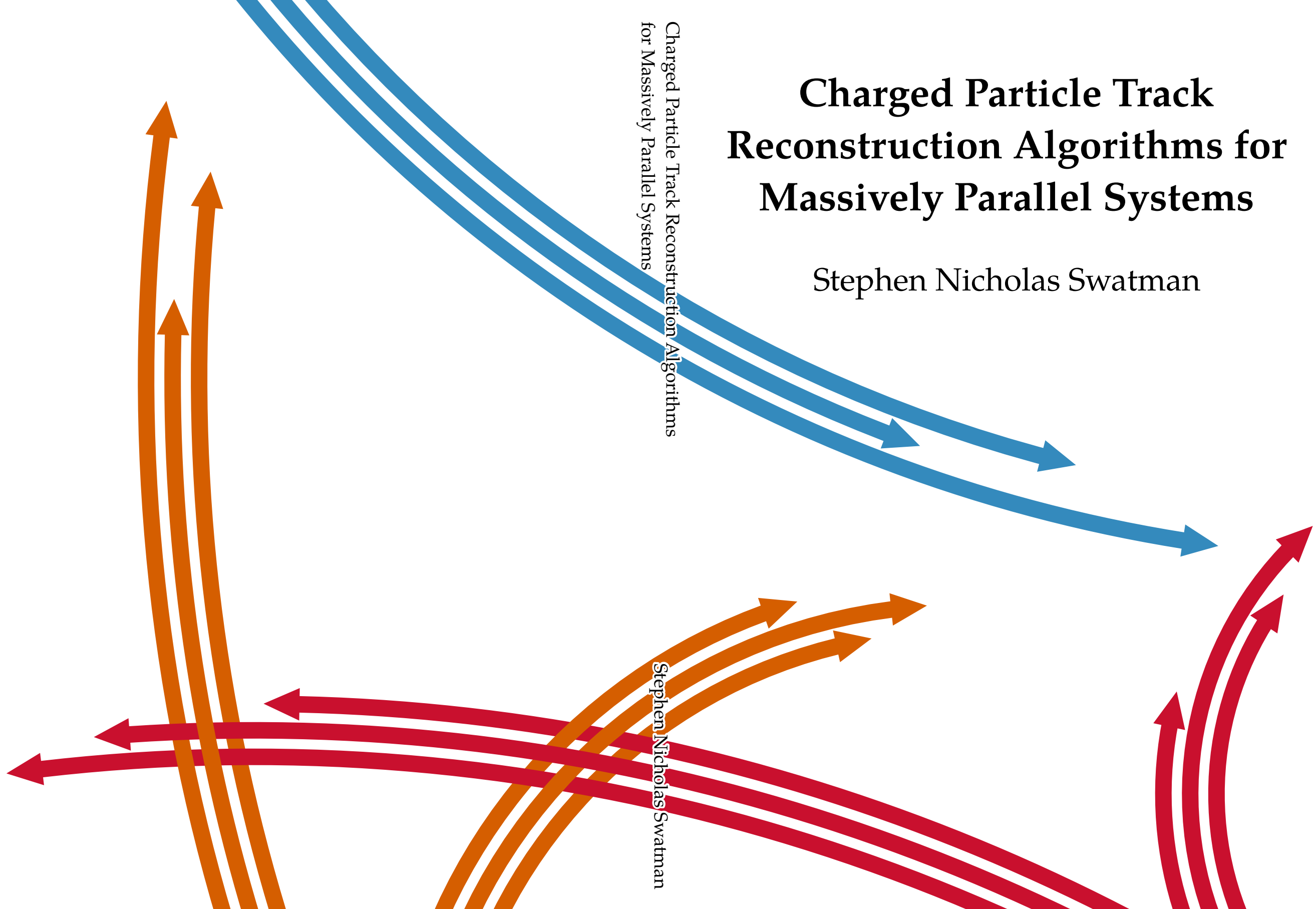


Charged Particle Track Reconstruction Algorithms for Massively Parallel Systems

Stephen Nicholas Swatman

Charged Particle Track Reconstruction Algorithms
for Massively Parallel Systems

Stephen Nicholas Swatman



Charged Particle Track Reconstruction Algorithms for Massively Parallel Systems

Stephen Nicholas Swatman



The work described in this thesis was carried out in the Parallel Computing Systems (PCS) group of the Institute for Informatics (IvI) at the University of Amsterdam, and within the CERN Doctoral Student Programme.

Copyright © 2025 Stephen Nicholas Swatman. Version d084349.

This thesis was typeset by the author using \LaTeX 2_ε and KOMA-Script. The cover of this thesis was designed by the author. This thesis was written by the author; no generative artificial intelligence was involved in the creation of any part of this work, including its text, its figures, and its cover. Physical copies of this thesis were printed and bound by Ipskamp Printing.

ISBN-13: 978-94-6473-879-7

Charged Particle Track Reconstruction Algorithms for Massively Parallel Systems

Academisch Proefschrift

ter verkrijging van de graad van doctor

aan de Universiteit van Amsterdam

op gezag van de Rector Magnificus

prof. dr. ir. P.P.C.C. Verbeek

ten overstaan van een door het College voor Promoties ingestelde commissie,

in het openbaar te verdedigen in de Agnietenkapel

op dinsdag 16 september 2025, te 13.00 uur

door

Stephen Nicholas Swatman

geboren te Amsterdam

Promotiecommissie

Promotores:	prof. dr. A.D. Pimentel	Universiteit van Amsterdam
	prof. dr. ir. A.L. Varbanescu	Universiteit Twente
Copromotores:	dr. A. Salzburger	CERN
Overige leden:	prof. dr. G.K. Keller	Utrecht University
	prof. dr. M. Weiland	University of Edinburgh
	prof. dr. P.T. Groth	Universiteit van Amsterdam
	prof. dr. R.V. van Nieuwpoort	Universiteit Leiden
	prof. dr. W. Verkerke	Universiteit van Amsterdam
	dr. S. Rehman	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

To the younger version of myself who was brave enough to take a plunge into the deep end, and to my parents, family, and friends who supported me every step of the way.

Summary

In this thesis, we investigate the feasibility of implementing algorithms for track reconstruction – the process of computing the continuous trajectories of charged particles in high-energy physics experiments according to the discrete measurements of those particles – on massively parallel architectures. Track reconstruction software commonly employs a task graph consisting of a variety of algorithms, including but not limited to algorithms inspired by computer vision, combinatorial algorithms, and numerical methods. Besides consisting of vastly different algorithms, track reconstruction software is also run in very different environments; on the one hand, so-called *offline* environments – in which the reconstruction of tracks takes place after they have been recorded to disk – require very high throughput. On the other hand, *online* environments – in which a decision is made whether to store or discard data immediately after it is gathered – present soft real-time constraints with latency requirements. The thesis includes a brief outline of high-energy physics in order to motivate our work in Chapter 2.

Although track reconstruction problems are currently solved using homogeneous CPU-based systems, such approaches will not be feasible for future experiments such as the Future Circular Collider (FCC), or even upgrades of existing experiments such the High-Luminosity Large Hadron Collider (HL-LHC); the amount of data to process would simply be too large. The field of high-energy physics is therefore looking towards massively parallel computing architectures – primarily General-Purpose Graphics Processing Units (GPGPUs) – and heterogeneous systems, which have been shown to provide higher performance, as well as higher energy efficiency, in a variety of scientific applications. Writing software for massively parallel architectures and for systems containing them is not easy, however: programming models vary between CPUs and GPUs, and optimisation strategies can differ strongly. As such, it is an open question to which extent massively parallel architectures can be applied to track reconstruction. We seek not only to understand *if* certain approaches work but also *why*, leading us to develop

Summary

novel models to better understand which factors promote and inhibit parallelism, not only in high-energy physics but in scientific computing in general. Furthermore, we aim to develop future-proof solutions: we expect our contributions to be relevant for the High-Luminosity Large Hadron Collider (HL-LHC) era, i.e. the next 17 years – and possibly beyond that. Therefore, we aim to develop methodologies and software that will be reusable and reproducible for the foreseeable future. The thesis includes an overview of high-performance computing Chapter 3.

We present an analysis of state-of-the-art track reconstruction software in Chapter 4, from which we identify two primary factors which inhibit the performance of track reconstruction software on massively parallel architectures. The first of these factors is the storage of large, multi-dimensional arrays. We address this topic in Chapter 5 of the thesis, based on the paper ‘Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Composition’. Such arrays are used principally to store magnetic fields, containing vector-valued data. Track reconstruction applications access such data very frequently and, as such, they can be critical performance bottlenecks. To complicate matters further, the most optimal way to store and access such data can change from device to device. For example, different CPUs may have different cache hierarchies, and the landscape of GPUs equipped with, e.g. texture units is even more complex. In this thesis, we present a novel method to allow for the systematic exploration of the design space of large multi-dimensional array storage. By decomposing the aforementioned design space into components, including but not limited to storage orders, interpolation methods, and boundary checking methods, we can find high-performance storage solutions for a broad range of devices automatically.

Extending our work on array storage, we find that there exists a family of multi-dimensional array layouts inspired by the so-called Morton layout. We discuss this family of layouts in Chapter 6 of the thesis, based on the paper ‘Using Evolutionary Algorithms to Find Cache-Friendly Generalized Morton Layouts for Arrays’. This family of array layouts is so large that it cannot be explored exhaustively. We show that this design space is not only *worth* exploring because it can yield significant performance benefits in real-world applications, but also that it *can* be efficiently explored using evolutionary algorithms. We develop a general method that can be applied to any algorithm or program using multi-dimensional arrays, and our method can increase performance in such applications without requiring any invasive modifications to the algorithm as compared to, e.g. matrix tiling.

The second performance factor we identify is *thread imbalance*, which we tackle in Chapter 7 of the thesis, based on the paper ‘Modelling Performance Loss due

to Thread Imbalance in Stochastic Variable-Length SIMT Workloads’. Although devices such as GPUs can achieve very high performance on regular applications, track reconstruction algorithms are often irregular, i.e. work cannot be evenly distributed across threads. Thread imbalance can be resolved or improved using techniques such as *thread coarsening* or *thread refinement*, but such strategies are hard to implement and their effects on performance can be hard to predict. In order to facilitate the use of coarsening and refinement methods, we develop a novel statistical model that can predict the performance impact of such approaches using information about the distribution of workloads *only*: the fact that our model is agnostic even to, e.g. the performance of existing implementations allows it to be used very early in the development cycle of parallel software, potentially saving significant amounts of time in the development of track reconstruction software and GPGPU programs more broadly.

To conclude the thesis, we present an implementation of track reconstruction for GPUs based on the lessons learned in the aforementioned chapters; this forms Chapter 8 of the thesis. Our work consists of novel implementations of all the components of the track reconstruction pipeline, and outperforms state-of-the-art algorithms on CPUs. We use specific techniques, based on our research on imbalanced execution to refine and coarsen the workload of threads in order to increase performance, and we employ specifically tuned array storage methods in order to improve access latency and throughput to large magnetic fields. Our work presents an important step not only in showing that massively parallel architectures are indeed feasible for solving the track reconstruction problem, but also in providing a sustainable and future-proof code base that can be used by high-energy physics experiments around the world.

Samenvatting

In dit proefschrift onderzoeken we implementaties van algoritmen voor spoorreconstructie – een proces in de hoge-energiefysica waar de continue trajecten van geladen deeltjes die een experiment doorkruisen worden gereconstrueerd aan de hand van discrete metingen – in massaal parallelle computerarchitecturen. De meeste spoorreconstructiesoftware bestaat uit een graaf van rekenkundige algoritmes, waaronder numerieke en combinatorische methoden. Naast het feit dat spoorreconstructie gebruik maakt van een aantal zeer verschillende algoritmen, is het ook belangrijk dat spoorreconstructiesoftware in een breed scala aan omgevingen ingezet kan worden; aan de ene kant bestaan er zogenaamde *offline* omgevingen waar data wordt verwerkt nadat deze zijn opgeslagen op een permanent opslagmedium. *Offline* omgevingen vragen om een zeer hoge verwerkingscapaciteit. Aan de andere kant bestaan er *online* omgevingen, waarin razendsnel tot beslissingen moet worden gekomen om data dan wel te bewaren, dan wel te verwerpen. Dergelijke *online* omgevingen vereisen juist zeer lage wachttijden. In Hoofdstuk 2 van dit proefschrift geven we een korte samenvatting van enkele basisprincipes uit de hoge-energiefysica.

Heden ten dage worden spoorreconstructieproblemen opgelost met behulp van homogene computersystemen gebaseerd op Central Processing Units (CPUs), maar deze aanpak zal hoogstwaarschijnlijk onvoldoende zijn om in de eisen van toekomstige experimenten te voorzien, zoals die van de Future Circular Collider (FCC), maar ook die van de geupgradete experimenten op de High-Luminosity Large Hadron Collider (HL-LHC). De hoeveelheden te verwerken data zullen voor huidige technieken simpelweg te groot zijn. Om deze uitdaging het hoofd te bieden zoeken fysici oplossingen in de vorm van massaal parallelle processorarchitecturen zoals zogenaamde General-Purpose Graphics Processing Units (GPGPUs) en heterogene systemen, waarvan is aangetoond dat ze een hoog prestatievermogen alsook een hoge efficiëntie kunnen behalen in bepaalde wetenschappelijke applicaties. Het schrijven van software voor zulke systemen is niet makkelijk: er

bestaan verscheidene programmeermodellen voor zowel CPUs als GPGPUs, en optimalisatiestrategieën kunnen sterk verschillen. Het is dan ook een open vraag of en in hoeverre het mogelijk is om spoorreconstructiesoftware te ontwikkelen voor massaal parallelle architecturen. In dit proefschrift pogen we aan te tonen dat dit inderdaad mogelijk is, alsook om nieuwe modellen te ontwikkelen om te begrijpen welke factoren parallellisme in dergelijke applicaties bevorderen of juist tegenwerken. Ons doel is om deze modellen en methoden toe te passen in de hoge-energiefysica maar ook om deze te generaliseren naar een breder scala aan applicaties. Daarnaast beogen wij in dit proefschrift toekomst-bestendinge oplossingen te ontwikkelen: het HL-LHC tijdperk zal de komende 17 jaar beslaan, en het is dan ook van belang om herbruikbare software te ontwikkelen en reproduceerbare resultaten te verzamelen. In Hoofdstuk 3 van dit proefschrift geven we een kort overzicht van massaal parallelle processorarchitecturen.

In Hoofdstuk 4 presenteren we een overzicht van het huidige spoorreconstructie-landschap, vanwaaruit we twee belangrijke effecten identificeren die de prestaties van massaal parallelle spoorreconstructiesoftware belemmeren. De eerste van die factoren is de opslag van grote, multi-dimensionale datareeksen. We behandelen dit onderwerp in Hoofdstuk 5 van dit proefschrift, gebaseerd op het artikel ‘Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Composition’ (‘Het Systematisch Verkennen van Representaties van Vectorvelden met Hoge Prestaties door middel van Compositie ten tijde van Compilatie’). Dergelijke multi-dimensionale reeksen worden in de hoge-energiefysica veelal gebruikt om magneetvelden op te slaan. Spoorreconstructiesoftware benötigt met een zeer hoge frequentie data uit deze reeksen en deze reeksen zijn daarmee belangrijke knelpunten voor de prestaties van de software als geheel. Verder kan de optimale opslagstrategie van processor tot processor verschillen. Verschillende CPUs hebben bijvoorbeeld verschillende *cache*-hierarchieën, en het GPU-landschap – gewapend met bijvoorbeeld *texture* processoren – is zelfs nog complexer. In dit proefschrift presenteren we een nieuwe methode die toestaat de ontwerpruimte van multi-dimensionale reeksen systematisch te verkennen. Door deze ontwerpruimte te deconstrueren tot verschillende opslagvolgorden, interpolatiemethoden, enz., kunnen we automatisch opslagstrategieën vinden die hoge prestaties bieden.

We breiden ons onderzoek naar de opslag van multi-dimensionale reeksen uit in Hoofdstuk 6, waar we een nieuwe ontwerpruimte voor opslagvolgorden voor dergelijke reeksen ontdekken. Deze volgorden zijn geïnspireerd door de zogenaamde Morton-volgorde. Het eerdergenoemde hoofdstuk is gebaseerd op het

artikel ‘Using Evolutionary Algorithms to Find Cache-Friendly Generalized Morton Layouts for Arrays’ (‘Het Vinden van Cache-Vriendelijke Gegeneraliseerde Morton-Opslagmethoden voor Multi-Dimensionale Reeksen met behulp van Evolutionaire Algoritmen’). Hoewel we aantonen dat het de moeite waard is om deze ontwerpruimte te verkennen omdat dit prestatiewinsten kan geven in verschillende applicaties, is de ontdekte ontwerpruimte dusdanig groot dat het uitputtend verkennen ervan onhaalbaar is. We tonen aan dat dit probleem verholpen kan worden door middel van evolutionaire algoritmen. We ontwikkelen een algemene methode die kan worden toegepast in een breed scala aan algoritmen waarin multi-dimensionale reeksen worden gebruikt, zonder dat het algoritme invasief aangepast hoeft te worden, zoals nodig is voor het toepassen van bekende optimalisatiestrategieën zoals het zogenaamde tegelen van matrices (*matrix tiling*).

Naast het opslaan van multi-dimensionale reeksen wenden we ons tot het concept van onbalans in massaal parallelle algoritmen in Hoofdstuk 7 van dit proefschrift, gebaseerd op het artikel ‘Modelling Performance Loss due to Thread Imbalance in Stochastic Variable-Length SIMT Workloads’ (‘Het Modelleren van Prestatieverlies Wegens Onbalans tussen *Threads* in Stochastische SIMT Operaties van Variabele Lengte’). Hoewel massaal parallelle apparaten als GPUs hoge prestaties kunnen waarmaken in reguliere applicaties waar de hoeveelheid werk tussen *threads* ongeveer gelijk is, hebben algoritmen in de spoorreconstructie juist vaak sterk uiteenlopende hoeveelheden werk, wat het moeilijk maakt om dit werk gelijk te verdelen over *threads*. Dergelijke onbalans tussen *threads* kan vaak (deels) worden verholpen met behulp van technieken zoals het verkleinen of juist vergroten van de hoeveelheid werk per *thread* – wat respectievelijk bekend staat als *thread refinement* en *thread coarsening* – maar dergelijke strategieën zijn vaak lastig te implementeren en de effecten ervan zijn lastig a priori in te schatten. Om het gebruik van dergelijke methoden te faciliteren, ontwikkelen we een nieuw, statistisch model dat de eerdergenoemde effecten op de prestaties van een programma kan voorspellen aan de hand van enkel informatie over de statistische distributie van de hoeveelheden werk. Ons model is agnostisch met betrekking tot bijvoorbeeld de bestaande prestaties van een algoritme en kan daarom zeer vroeg in het ontwikkelingsproces van parallelle software worden toegepast, wat mogelijk kan leiden tot significante tijdsbesparing.

Ter afsluiting van dit proefschrift presenteren we in Hoofdstuk 8 een implementatie van spoorreconstructie voor GPUs gebaseerd op de lessen die uit de eerdere hoofdstukken van het proefschrift zijn getrokken. Ons werk bestaat uit nieuwe implementaties van alle componenten van het spoorreconstructieproces

Samenvatting

en presteert beter dan algoritmen ontwikkeld voor CPUs. We maken gebruik van specifieke technieken gebaseerd op het vergroten of verkleinen van de hoeveelheid werk per *thread* om prestaties te winnen, en we maken gebruik van specifiek ontwikkelde opslagmethoden voor multi-dimensionale reeksen om de toegangssnelheid van het geheugen te verbeteren voor grote magneetvelden. Ons werk is een belangrijke step in het aantonen dat massaal parallelle processorarchitecturen bruikbaar zijn voor het spoorreconstructieprobleem, alsook een belangrijke stap in het ontwikkelen van software die van belang kan zijn voor experimenten in de hoge-energiefysica over de hele wereld, vandaag en in de toekomst.

Contents

Summary	vii
Samenvatting	xi
1 Introduction	1
1.1 Observing the Infinitesimally Small	2
1.2 Research Questions	5
1.3 Thesis Outline	7
2 Particle Physics and Track Reconstruction	9
2.1 Particle Physics and the Standard Model	9
2.2 Accelerator and Collider Physics	12
2.3 The <i>Large Hadron Collider</i>	14
2.4 Particle Detectors and Experiments	18
3 Parallel and Massively Parallel Computing	25
3.1 History of Hardware Performance	25
3.2 Parallel Computing	28
3.3 Graphics Processing Units	34
3.4 Programming Models	44
4 State-of-the-Art Algorithms for Track Reconstruction	47
4.1 Introduction	47
4.2 Existing Implementations and Challenges	49
4.3 Preprocessing	53
4.4 Track Finding	58
4.5 Track Refinement	68
4.6 Summary	72

5	Exploring the Design Space of Vector Field Representations	75
5.1	Introduction	76
5.2	Background	77
5.3	Design Space Exploration	79
5.4	Access Patterns	81
5.5	Storage Backends	84
5.6	Evaluation	96
5.7	Applicability and Limitations	102
5.8	Reproducibility and Reusability	104
5.9	Summary	104
6	Finding Cache-Friendly Data Layouts Through Evolution	107
6.1	Introduction	108
6.2	Background and Related Work	109
6.3	Generalised Morton Layouts	113
6.4	Exploration Through Evolution	120
6.5	Evaluation	122
6.6	Limitations and Threats to Validity	131
6.7	Reproducibility and Reusability	132
6.8	Summary	132
7	Modelling and Mitigating Thread Imbalance in SIMT Workloads	135
7.1	Introduction	136
7.2	Background	137
7.3	Mitigating Imbalance	139
7.4	Modelling Imbalance	141
7.5	Model Evaluation	149
7.6	Practical Implications	157
7.7	Related Work	159
7.8	Reproducibility and Reusability	160
7.9	Summary	160
8	Evaluating the Viability of Massively Parallel Track Reconstruction	161
8.1	Preprocessing	162
8.2	Track Finding	168
8.3	Track Refinement	178
8.4	Task Graph Performance Bounds	179
8.5	Results	190

8.6	Reproducibility and Reusability	193
8.7	Summary	195
9	Conclusion	199
9.1	Research Questions	200
9.2	Outlook and Future Work	203
9.3	Final Thoughts	204
	Acknowledgements	207
	List of Publications	211
	List of Acronyms	215
	Bibliography	219

1

Introduction

*I know it's true;
it's all because of you.
And if I make it through;
it's all because of you.*

— John Lennon
(Musician)

As human beings, it is in our nature to be curious about the world around us. For thousands of years our ancestors have looked up at the night sky and wondered about some of the largest structures in our universe. Over time, we have come to understand that the planet on which we live is part of the *Solar System*, which in turn is part of the *Milky Way Galaxy*. The Milky Way itself is but one part of the so-called *Local Group*, contained within the *Virgo Supercluster*, one of approximately ten million superclusters in the observable universe [4]. The scale of the universe in which we live is so massive that it is almost incomprehensible, yet humanity's insatiable thirst for knowledge has driven centuries of research into *outer space*.

Similarly, our inquisitive nature extends to *inner space*. Philosophers of ancient times – Greek, Indian, and Tibetan alike – considered the constituents of matter in an attempt to understand the world not on the scale of the unfathomably large, but rather on a scale of the infinitesimally small. Although the Empedoclean elements of fire, air, water, and earth are considered obsolete in modern science [5], Leucippus' contemporary idea of atomism gave rise to more than two thousand years of research into the fundamental constituents of matter [6], a field which we now refer to as *particle physics*. Over time, humanity discovered that life was composed of *cells*. Cells – as well as most other matter – were subsequently found to be made up of *molecules*. In an act of hubris, the building blocks of molecules became known as *atoms*, from the Greek word for indivisible, only to be found to contain even smaller particles no more than a hundred years later. Today, we know that even the *nucleons* that make up atoms are divisible into even smaller

particles known as *quarks* [7].

The development of the sciences of the infinitely large and the infinitesimally small were both driven by the development of new tools and methods. Astronomy advanced greatly with the invention of the optical telescope, and advanced radio telescopes drive research today. Microbiology was made possible by the invention of the microscope, and modern particle physics relies on highly advanced methods of particle detection. Undoubtedly the largest drivers of innovation across all sciences, however, have been the advancement of mathematics and the development of the computer: developments in all fields of mathematics have given rise to new scientific methodologies and modern computers have turned many of these developments from abstract ideas to concrete tools for research [8]. The ability of mechanical and electrical minds to perform rote calculations dwarfs human ability to the same degree that the Milky Way dwarfs a human in size¹, and they have enabled the development of scientific simulations, experiments, and analyses on scales never before seen.

Despite the tremendous advances in the fields of particle physics and computing, the human desire for knowledge seems insatiable. As David Hilbert famously said: ‘We must know. We shall know.’ We must know more about the structure of our universe and about the nature of matter, and to know more, we must compute more. This thesis is our contribution, however small, to humanity’s ability to more effectively and efficiently use computers to understand the world around us.

1.1 Observing the Infinitesimally Small

In the late seventeenth century, Antoni van Leeuwenhoek studied the cells that made up living tissue using a microscope of his own design [10]. Van Leeuwenhoek’s microscope provided him with magnification of up to 270 times, allowing him to observe a variety of microscopic processes with his own eyes. The idea observing a phenomenon directly, i.e. to be an eye witness of it, is certainly a romantic one, but it is ultimately infeasible when we enter the world of the very small. Optical instruments, no matter how precisely manufactured, will eventually fail to resolve sufficiently small objects due to the effects of diffraction [11]. Thankfully, human ingenuity has allowed us to use *indirect* observation to make sense of things

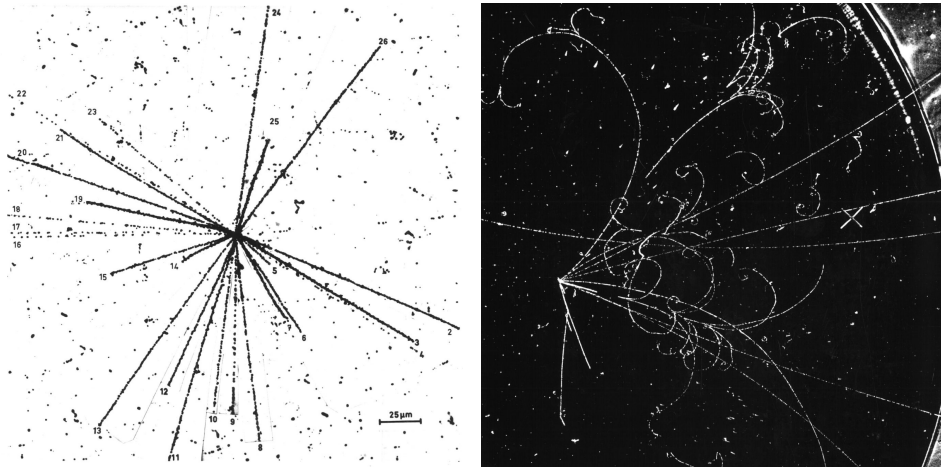
¹*Frontier*, the faster supercomputer in the world at the time of writing, is capable of performing 1.102 quintillion floating point operations per second [9]; if we generously assume that a human can perform one such operation per minute, *Frontier* beats the human brain by twenty orders of magnitude. Similarly, the diameter of the Milky Way is approximately twenty orders of magnitude larger than the height of the average human.

which we cannot observe *directly*.

A prime example of indirect observation is the so-called Rutherford experiment. In the early twentieth century, scientists were divided on the structure of the atom. Two prevailing theories were the Thomson model, which stated that the atom consisted of a positively charged ‘plum pudding’ with negatively charged raisins strewn throughout, while the Rutherford model claimed that the atom consisted of a dense positively charged nucleus with negatively charged particles orbiting around it. In order to test these hypotheses, Hans Geiger and Ernest Marsden – under the supervision of Ernest Rutherford – shot alpha particles at a thin golden foil. They posited that the way the particles would interact with the gold atoms would reveal the structure of the atom; if the Thomson model were correct, the particles would fly straight through the atom, and if the Rutherford model were correct, the particles would bounce off the nucleus. The results they collected confirmed the Rutherford model [12], but more pertinently, virtually nothing in this experiment could be directly observed: the structure of the gold atoms was obviously too small to observe optically, but even the alpha particles with which they bombarded the gold were so tiny that they needed to be observed indirectly².

The indirect observation of charged particles – including alpha particles – is an important topic in particle physics. Indeed, it is what allowed Geiger and Marsden to conduct the Rutherford experiment, and it also allows us to observe neutral particles – through their decays or interactions with matter – which would otherwise be almost impossible to detect at all. Thankfully, the fact that charged particles are, as their name suggests, electrically charged makes them eager to interact with other kinds of matter. In the Rutherford experiments, alpha particles were detected using a fluorescent screen which would produce a small flash of light when a charged particle passed through it [12]. Other common methods for detecting charged particles and their trajectories are the use of so-called *nuclear emulsion* – particles of silver suspended in a medium, usually gelatin – which would act somewhat like a photographic plate [15] and the use of *bubble chambers* – chambers filled with barely sub-critical gas which forms bubbles upon interaction with a charged particle – which could be photographed [16]. Recordings of charged particles using nuclear emulsion and a bubble chamber are shown in Figures 1.1a

²It is noteworthy that the art of indirect observation comes quite naturally to humans. At Nikhef, the Dutch national institute for particle physics, a popular attraction for the general public is an experiment which emulates the Rutherford experiment. In the middle of an inclined wooden surface, we place an object of arbitrary shape, e.g. a triangle or a circle. The shape of the object is hidden from the audience, which is invited to roll ball bearings down the board. The vast majority of people, including children, is able to correctly identify the shape of the concealed object simply by observing how the ball bearings are deflected by the hidden object.



(a) Microphotograph a plate of nuclear emulsion capturing the disintegration of an atomic nucleus [13]. (b) Photograph of a bubble chamber capturing a high-energy interaction produced by a neutrino [14].

Figure 1.1: Two examples of analogue particle detection methods. Note that the trajectories of the particles can be traced (nearly) continuously. Both images provided by CERN.

and 1.1b, respectively.

Figure 1.1 shows how beautifully and clearly analogue particle detection methods mark the trajectories of particles. The unfortunate downside of these methods is that they allow relatively low rates of data collection; nuclear emulsions need to be developed, and bubble chambers require time for the bubbles to dissipate. Like many fields of modern science, particle physics relies on large volumes of data to make statistically significant claims, and – as we will see in Chapter 2 – taking data at the requisite rates using analogue methods is no longer feasible. Many modern particle physics rely on silicon-based particle detectors, not dissimilar to the photographic sensors in modern digital cameras [17]. Silicon-based detectors allow for tremendously high rates of data collection, but they are unable to record the trajectories of particles in the (nearly) continuous way that nuclear emulsions and bubble chambers can; instead, they allow for measurements only in predetermined, discrete locations.

We have now arrived at the *track reconstruction* problem: given a set of discrete measurements of one or more charged particles, what was the true trajectory of each of those particles. The track reconstruction problem – an advanced form

of connecting-the-dots – is a crucial part of modern particle physics, and it requires a significant amount of computational effort to solve. The computational requirements for track reconstruction are so great, in fact, that current solutions are unlikely to scale to future particle physics experiments [18, 19]. Therefore, particle physicists are now looking to massively parallel processor architectures such as General-Purpose Graphics Processing Units (GPGPUs) in order to meet the computing challenges of the future. GPGPUs have been applied to many other problems in particle physics, e.g. statistical computations [20], but massively parallel track reconstruction remains an open challenge.

1.2 Research Questions

In this thesis, we work towards the development of track reconstruction software that can meet the challenges posed by the increased data volumes of future colliders and experiments in high energy physics. In particular, we aim to evaluate the feasibility of implementing such software on massively parallel systems such as general-purpose graphics processing units. Furthermore, we aim to develop novel algorithms and methods to enable further developments in the field of high-performance track reconstruction. The overarching research question for this thesis, therefore, is the following:

‘Can track reconstruction be implemented efficiently on massively parallel systems?’ (MRQ)

In order to answer the Main Research Question, it is fundamental to understand the state of the art in the field of track reconstruction and to understand the shortcomings of existing work when it comes to high-throughput computing as well as execution in massively parallel environments. We aim to develop a deep understanding the so-called extra-functional properties of existing algorithms and implementation through empirical methods such as profiling as well as through descriptive and predictive models. Our second research question, therefore, is the following:

‘What are the challenges in porting state-of-the-art reconstruction algorithms to massively parallel architectures?’ (RQ1)

Answering Research Question 1, we find that data structures representing magnetic fields, i.e. vector fields, are an important consideration in track reconstruction. More generally, such structured grid data is an important element in many kinds of

computation. Indeed, it is considered one of thirteen *dwarves* of parallel computing: basic patterns of computation from which most if not all modern parallel kernels are composed [21]. The implementation design space for structured grid data – e.g. data layouts, interpolation methods, boundary handling methods, etc. – is very large. We therefore aim to automatically explore this design space in order to maximise the efficiency of hardware caches and, consequentially, the performance of software. This leads us to our third research question:

‘How can structured grid data be represented in order to maximally exploit the access locality of arbitrary computations, including those in track reconstruction?’ (RQ2)

We also find that kernels in track reconstruction are highly irregular in terms of workload. This is not a problem for traditional CPU architectures, but threatens the performance of massively parallel GPGPU architectures through the introduction of thread imbalance. Although the effects of imbalance are perhaps higher in track reconstruction kernels than they are in most other high-performance computing applications, thread imbalance is an issue encountered across many fields of computational science. It is, therefore, useful to be understand, model, and mitigate the effects of thread imbalance. Our fourth research question, thus, is as follows:

‘How can the effects of thread imbalance in SIMT workloads be modelled and how can they be mitigated?’ (RQ3)

Research Questions 2 and 3 tackle some of the primary obstacles in developing massively parallel track reconstruction software. Using our novel models and methods, we can implement a massively parallel track reconstruction software package. In order to assess the efficacy of this software package, we posit our fifth and final research question:

‘How do the extra-functional properties of novel track reconstruction algorithms, designed to exploit massive parallelism, compare to state-of-the-art solutions?’ (RQ4)

An overarching theme for the work presented in this thesis is reproducibility and reusability. The field of high-energy physics operates on long timescales; the Large Hadron Collider – an important experiment in the field – is expected to operate until at least the end of 2041, i.e. for the coming seventeen years. As such, we believe it is important for the models and methods presented in this thesis to stand the test of time, and for them to be reused, reproduced, and improved upon in future research. For this reason, we have endeavoured – to the best of our

ability – to provide permanently archived software artifacts to go along with our research and we have, wherever possible, submitted these artifacts for review by the scientific community. Wherever relevant, we will highlight these artifacts and their availability.

1.3 Thesis Outline

A schematic overview of the chapters in this thesis, as well as the ways in which they connect, is shown in Figure 1.2. In Chapters 2 and 3 we cover the relevant background information in high-energy physics and high-performance computing, respectively. We provide an overview of the algorithms used in state-of-the-art track reconstruction software in Chapter 4, where we also develop analytical performance models for the track reconstruction problem. In Chapter 5 we propose and implement a novel method for exploring the design space of structured grid data. We expand on Chapter 5 by examining a generalisation of the so-called Morton storage order in Chapter 6 and by employing evolutionary algorithms to automatically find cache-friendly array layouts. Chapter 7 proposes a model for evaluating the performance of stochastically imbalanced workloads on Single Instruction Multiple Data (SIMD) and Single Instruction Multiple Threads (SIMT) architectures, which are common in track reconstruction software. We combine our findings to develop a new chain of algorithms, optimised for massively parallel execution, and evaluate the performance of this novel software in Chapter 8. Finally, we posit our conclusions and answer our research questions in Chapter 9.

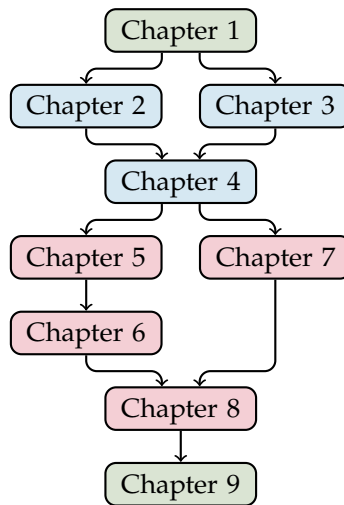


Figure 1.2: Schematic overview of the chapters in this thesis. Green chapters are the introduction and conclusion of our this thesis, blue chapters provide background information, and red chapters contain our scientific contributions.

2

Particle Physics and Track Reconstruction

It is often easier to ask for forgiveness than it is to ask for permission.

— **Grace Hopper**
(Professor of mathematics)

The application considered in this thesis stems from the domain of particle physics. We look at this application from a strongly computational lens rather than a physical one, and as such no understanding of particle physics is strictly required to understand the work described in this thesis. We do, however, want to provide the reader with a surface-level understanding of some concepts in particle physics, as these will help to motivate *why* we are writing this thesis. This chapter, therefore, serves as a rudimentary introduction to the field of particle physics – intended primarily for an audience coming from the field of computer science – and the track reconstruction problem, for which we will attempt to develop high-performance solutions.

2.1 Particle Physics and the Standard Model

Particle physics is the study of the fundamental building blocks of matter in our universe, as well as of the forces which dictate the interplay between those building blocks. The matter that exists around us in our everyday life consists of molecules which – in turn – consist of atoms. Atom, then, can be further subdivided into electrons and nucleons – protons and neutrons. Nucleons are themselves composite particles and consist of quarks and gluons. Much of our current knowledge of the behaviour of the particles in our universe, as well as the forces that operate on those particles, is described in the so-called *Standard Model of Particle Physics*,

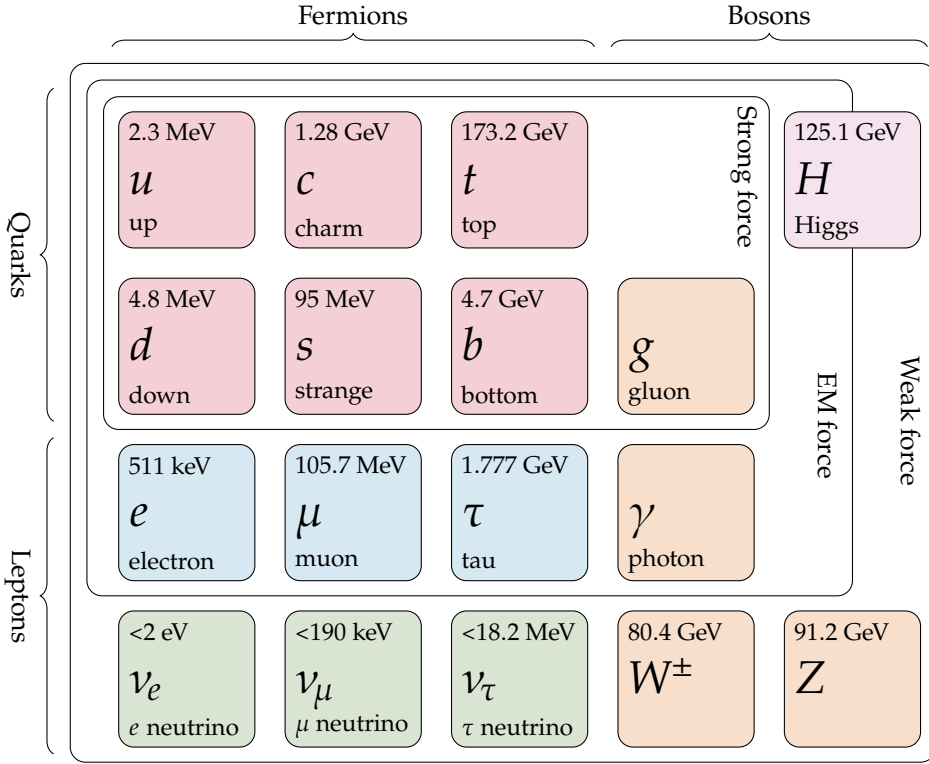


Figure 2.1: A simplified schematic overview of the particles in the standard model of particle physics. This overview does not include antiparticles. Listed above each particle are the measured masses of those particles. Graphic adapted from Burgard [22].

more commonly referred to simply as the *Standard Model*. The Standard Model is a complex mathematical model with a broad range of implications, but it is often represented as a table of particles like in Figure 2.1. Note that the Standard Model extends far beyond this list of particles, as it also describes, e.g. their interactions and decay modes, but these details are outside the scope of this thesis.

The Standard Model clearly divides particles into separate families. *Fermions*, defined as particles with half-integer *spin*¹, are the building blocks of matter. The family of fermions includes both quarks and leptons; the primary difference between these families of particles is that quarks interact with the so-called strong nuclear force, while leptons do not. In the ‘normal’ matter that surrounds us in our

¹The definition of *spin* falls well outside the scope of this thesis; it is a quantum-mechanical property of particles which is only loosely related to the concept of angular momentum as we understand it in the macroscopic world [23].

everyday lives, this means that quarks bundle together to form so-called *hadrons*, a category which includes the proton (composed of two up quarks and one down quark) and the neutron (composed of two down quarks and one up quark). Protons and neutrons constitute the nuclei of atoms.

Bosons, contrary to fermions, have integer spin and mediate the forces between fermions. As an example, the photon – perhaps the best known boson – is responsible for mediating the electromagnetic force. When two charged particles interact and either attract or repel each other by means of the electromagnetic force, this interaction is carried by photons, and it is these photons that cause the two particles to change course². Other bosons include the gluon which mediates the strong nuclear force, the W and Z bosons which mediate the weak nuclear force, and the Higgs boson which – through the so-called Higgs mechanism – imparts mass to other particles [23].

The Standard Model of Particle Physics has a remarkable history of predicting discoveries. In particular, the Standard Model predicted not only the existence but also several properties of the top quark, the tau neutrino, and the Higgs boson, which were experimentally confirmed in 1995 [24], 2000 [25], and 2012 [26, 27], respectively. The Standard Model is not complete, however. In particular, the Standard Model fails to explain the inner workings of gravity: out of the four known fundamental forces – the weak nuclear force, the strong nuclear force, electromagnetism, and gravity – only the first three are explained by the Standard Model, and gravity is not. Additionally, the Standard Model does little to explain the existence of *dark matter*: a hypothetical form of matter necessitated by the fact that we observe gravitational effects in our universe which are not fully explained by normal matter [28]. There are many theories describing so-called Beyond Standard Model (BSM) physics – which includes but is not limited to gravity and dark matter – but experimental evidence is necessary to confirm these theories. Additionally, the Standard Model contains a significant number of free parameters whose values can be refined by additional experimental evidence. In short, the Standard Model is simultaneously an extraordinarily accurate model of our universe, but also incomplete and in need of further refinement, and it is through *collider physics* that particle physicists are looking to tackle these challenges.

²It is important to state that this explanation, is a significant simplification. The electromagnetic force is carried by so-called *virtual* photons, which cannot be understood as particles per se.

2.2 Accelerator and Collider Physics

It is somewhat rare for equations to embed themselves into popular culture to the same degree that, say, a popular song lyric would, but there is one equation which has managed to capture the imagination of so many that it has arguably taken on an ambassadorial role for physics and science as a whole. In 1905, Albert Einstein – himself an icon of science in our time – described the equivalence between mass and energy [29], which would later – somewhat inaccurately – become famous as the equation $E = mc^2$. Briefly, this equation establishes an equivalence between matter and energy: a very small amount of matter can be turned into a very large amount of energy, and a very large amount of energy can consequently also be turned into a very small amount of matter. It is the latter part of this idea that motivates accelerator and collider physics.

The Standard Model describes many unstable particles, including the second and third generation fermions and leptons as well as hadrons composed of the aforementioned. These unstable particles are heavy and decay into lighter, stable particles over time according to the mass-energy equivalence described earlier. By virtue of the universe being billions of years old, virtually all such particles have decayed into the ordinary matter (mostly up quarks, down quarks, and electrons) that surrounds us in our day to day life. To borrow terminology from the field of high-performance computing, the universe has entered a sort of *steady state*. Thankfully, unstable particles can still be created and – as such – it can still be studied; all we need to do is to gather enough energy in a sufficiently small amount of space that, by Einstein's equation, matter can be created from energy. The key intuition here is that to create a particle with a given mass, one has to have at least the energy equivalent of that mass available. Classically, high energy particles coming from space (e.g. from the Sun) can be used to study the properties of unstable particles, but these so-called *cosmic rays* are hard to predict and harder to control.

Thankfully, humans have learned how to harness energy for the use of particle physics experiments through the development of *particle accelerators*. By imparting kinetic energy to particles, they can be used in high-energy physics experiments. Indeed, nature does not discriminate between energy that particles gain by being created inside of a star and energy that a particle might gain by passing through a strong human-made electric field. Particle accelerators are generally either linear, in which a particle experiences a fixed series of accelerations along a straight trajectory, or circular, in which particles can circle around arbitrarily many times,

acquiring more and more energy over time. Both linear and circular colliders have advantages and disadvantages [30, 31], and the methods discussed in this thesis are relevant to both designs.

It is worth noting that it is not conventional to measure the result of acceleration in particle physics – i.e. velocity – in meters per second (or equivalent units) like we do in the macroscopic world. In our day-to-day lives, the velocity of an object grows according to the square root of its kinetic energy: quadrupling the kinetic energy in an object doubles its velocity. Clearly, this idea is incompatible with the widely understood fact that nothing can travel faster than the speed of light: given a sufficiently large amount of energy, we would eventually be able to accelerate an object past the speed of light. In reality, the classical equations of motion are simply a low-energy approximation of the equations of motion governed by general relativity: as we impart more energy to a particle, its velocity asymptotically approaches the speed of light. This ensures that we can keep adding energy to a particle without breaking the laws of physics, but it also complicates conversations about velocity in a classical sense. In order to avoid this confusion, the convention in physics – to which we will also adhere in this thesis – is to measure the velocity – or, more precisely, the momentum – of a particle as its kinetic energy directly. The common unit for this is the *electronvolt* (eV). To convert between velocity as we know it in our day-to-day lives, we compute the so-called Lorentz factor γ based on the total energy of the particle (including the energy-equivalent of its rest mass and its kinetic energy) E_{total} and the rest mass energy-equivalent E_{rest} as in Equation 2.1 [32]:

$$\gamma = \frac{E_{\text{total}}}{E_{\text{rest}}} \quad (2.1)$$

We can then compute the fraction of the speed of light at which the particle is travelling, also referred to as the β factor, according to Equation 2.2:

$$\beta = \frac{v}{c} = \sqrt{1 - \left(\frac{1}{\gamma}\right)^2} \quad (2.2)$$

Regardless of how we express the velocity or energy of particles, the primary objective of an accelerator is to impart to them as much energy as possible. This energy can be used to create new particles by shooting them at a fixed target (a so-called *fixed-target experiment*), but this is rather inefficient. Indeed, the conservation of momentum dictates that the particles produced in such a collision would have significant momenta of their own, leaving less energy for the creation of new

matter. The energy available for the creation of new matter is usually referred to as the centre-of-mass energy, and is often written \sqrt{s} . Because the momenta of two particles travelling in opposite directions cancel out, a collision between such particles provides a much higher centre-of-mass energy than a fixed target experiment [33]. Consequently, most modern particle physics experiment are based on particles travelling in opposite directions, and these are often referred to as *collider experiments*.

2.3 The *Large Hadron Collider*

At the time of writing, the largest and most powerful particle accelerator and collider in operation is the Large Hadron Collider (LHC). The LHC is operated by the European Organisation for Nuclear Research (CERN) and is located along the Swiss-French border, near Geneva. The Large Hadron Collider is a circular collider with a circumference of approximately 27 km. This collider maintains two beams, each with a maximum energy – at the time of writing – of 7 TeV, giving a centre-of-mass energy of $\sqrt{s} = 14$ TeV. The LHC is capable of accelerating protons as well as heavy ions (most commonly lead, but this can also include oxygen and other elements), but this thesis will focus strictly on proton–proton collisions, often referred to simply as *proton physics* [34].

In order to collide protons in the LHC, a complex chain of events has to take place. First, protons are drawn from a source of ordinary matter; in the case of the LHC, this is a canister of hydrogen gas^{3,4}. This hydrogen gas is ionised – i.e. it gains an additional electron and becomes negatively charged – after which it can be accelerated. In order to facilitate the operation of the LHC, protons are accelerated in smaller accelerators first before they enter the main beams; these smaller accelerators are referred to as pre-accelerators and act as metaphorical on-ramps for the LHC. The ionised hydrogen atoms are accelerated using Linac4, a linear accelerator, and reach an energy of 160 MeV. At this point, the anion with a rest mass of approximately 940 MeV has a total energy of 1100 MeV [35], giving – by Equation 2.1 – $\gamma \approx 1100/940 \approx 1.17$ and – by Equation 2.2 – $\beta \approx \sqrt{1 - (1/1.17)^2} \approx 0.51$. In other words, the anion is travelling at approximately half the speed of light.

Following the acceleration in Linac4, the hydrogen anion is stripped of both its electrons, leaving only a proton. This proton then enters the Proton Synchrotron

³We recall that a hydrogen atom consists of a single proton and a single electron.

⁴Although the LHC collides a very large number of protons, the total mass of these protons remains minuscule: during its predicted operation, the LHC will accelerate fewer protons than there are in a cup of espresso.

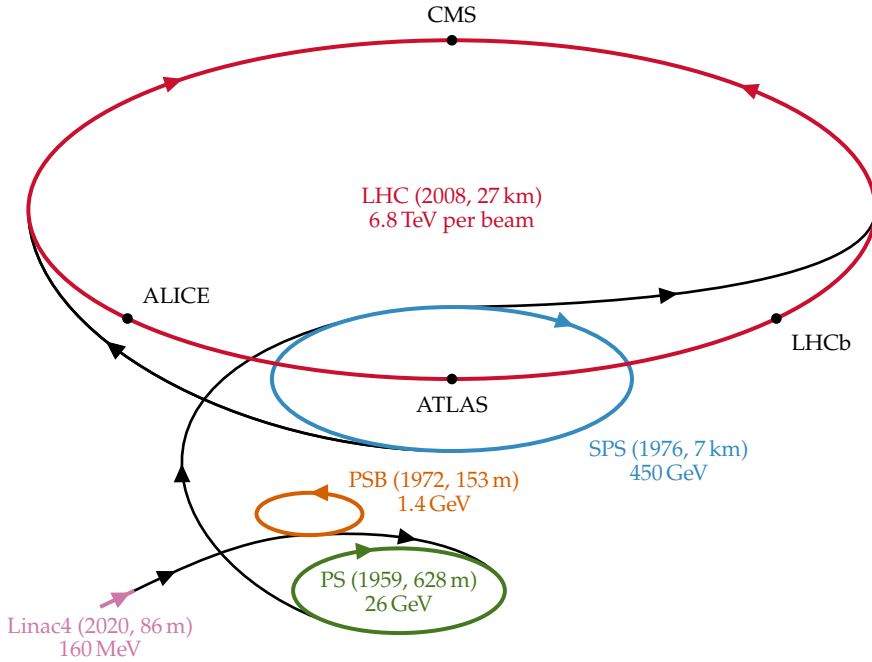


Figure 2.2: A schematic overview of the CERN accelerator complex, including the Large Hadron Collider (LHC) and its experiments, as well as the various pre-accelerators. Graphic adapted from Gessinger-Befurt [36].

Booster (PSB) – a circular collider 157 m in circumference – which accelerates it up to an energy of 2 GeV [37]. From there, the proton enters the Proton Synchrotron (PS) – another circular collider with a length of 628 m – for further acceleration up to 25 GeV [38]. Notably, the PS was completed in 1959 and was, at the time, the most powerful particle accelerator in the world; the fact that the PS is still operational to this day and now serves as a pre-accelerator for much more advanced colliders is a testament to the resourcefulness of particle physicists. Protons are inserted from the PS into the so-called SPS which has a length of around 7 km and which accelerates protons up to 450 GeV [39]. A schematic overview of the LHC accelerator complex is given in Figure 2.2.

It is worth noting that the LHC contains a great number of protons, grouped into so-called *bunches*. At any given time, up to approximately 2500 bunches travel around the collider in each direction⁵. At peak intensity, each bunch contains approximately 10^{11} protons [40], and this number gradually decreases as the bunches

⁵At a total length of 26 659 m and a bunch spacing of 25 ns [40], the LHC could hold a theoretical maximum of 3557 bunches, but technical limitations prohibit filling the collider fully.

go around the collider, principally due to the loss of protons in collisions. As the intensity of beams decreases, so does the viability of using them for physics; therefore, the beams are emptied in so-called *beam-dumps* when the intensity drops below a given threshold to make space for new beams. The entire process of injecting protons into the LHC, accelerating them to their highest energies, colliding them, and dumping them is referred to as a *fill*, and each fill takes approximately half a day, although fills can be much longer or much shorter than that. Proton physics is performed day and night in the LHC for around six months per year.

The large number of protons that circulate in the LHC is both a necessity and a curse. Indeed, having so many protons is necessarily because – practically speaking – colliding two individual protons head on is nearly impossible due to their small size, and due to the fact that we can only control their position with limited precision, i.e. they are positioned across a (usually multi-Gaussian) distribution – referred to as the *beam spot* – rather than in a single point. The effect of proton counts and beam spot size is clearly demonstrated by the calculation of the so-called *luminosity* \mathcal{L} of the Large Hadron Collider: a metric that describes roughly how many particles can be made to collide. Under the assumption that the collisions happen head-on and that the beams have a multi-Gaussian shape described by the standard deviations σ_x and σ_y , luminosity is calculated as in Equation 2.3, where f additionally denotes the crossing frequency and n_1 and n_2 denote the number of particles in the two beams [23]:

$$\mathcal{L} = f \frac{n_1 n_2}{4\pi\sigma_x\sigma_y} \quad (2.3)$$

For the Large Hadron Collider, $f \approx 40$ MHz, $n_1 \approx n_2 \approx 10^{11}$, and $\sigma_x \approx \sigma_y \approx 7 \mu\text{m}$ [41]. Equation 2.3 is an optimistic estimate, and the true luminosity is often lower; for the Large Hadron Collider, the realistic design luminosity amounts to $\mathcal{L} \approx 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$. It is worth noting that luminosity does not describe a rate of collisions directly, but rather a rate for a given area. The intuition here is that the area or cross-section of the colliding particles also impacts the rate of collisions: as a visual metaphor, one might imagine that it is easier to get two streams of tennis balls to collide than it might be to do so with two similarly packed streams of ping-pong balls. The rate of a specific type of a process i with cross-section σ_i is calculated from the luminosity as in Equation 2.4:

$$\frac{dN_i}{dt} = \sigma_i \mathcal{L} \quad (2.4)$$

The total number of events N with a cross-section σ_i is calculated according to Equation 2.5:

$$N_i = \int \mathcal{L}_i(t) dt \quad (2.5)$$

Finally, it is also customary to integrate the luminosity with respect to time – without incorporating the cross-section of a specific process – which gives a metric of the total amount of data gathered by an experiment. This is often referred to as the *integrated luminosity* and is given, perhaps unintuitively, in terms of inverse area. For example, in the 2022 and 2023 operating period the ATLAS experiment has gathered about $6.6 \times 10^{40} \text{ cm}^{-2}$ of integrated luminosity [42].

Since the cross-sections encountered in particle physics are usually very small, the convention is to use the *barn* unit of area – as in, ‘to hit the broad side of a barn’ – where $1 \text{ fb} = 10^{-24} \text{ cm}^2$. The cross-section of inelastic proton–proton collisions – that is to say, collisions in which the full kinetic energy is available for the production of new particles – is about 60 mb [43] and therefore it follows from Equation 2.4 that a Large Hadron Collider experiment can – according to the design luminosity – expect a proton–proton collision rate of approximately $\frac{dN_{pp}}{dt} = 60 \text{ mb} \cdot 10^{34} \text{ cm}^{-2} \text{ s}^{-1} = 600 \text{ MHz}$. With a sufficient amount of hand-waving, we can interpret the cross-section of a process as the probability of that process happening, expressed as an area.

We are now ready to return to our original argument as to why it is necessary to have such high collision rates in the Large Hadron Collider, and why it is important to have so many protons circling around at once. Simply put, many of the interesting processes in high-energy physics are very rare. Two processes which allow us to detect the sought-after Higgs boson, namely the production of a Higgs boson and its subsequent decay into two photons ($H \rightarrow \gamma\gamma$) and the production of a Higgs boson and its subsequent decay into four leptons via two Z bosons ($H \rightarrow ZZ^* \rightarrow 4\ell$), have a total cross-section of about 57 pb at the LHC energy scale [44]. It follows that an LHC experiment can expect to see such processes at a rate of 0.57 Hz; since a large number of such events must be observed to provide statistically significant evidence, high luminosities are critical in high-energy physics. As collision rates and bunch sizes directly impact luminosity, increasing these metrics is crucial to gather enough data.

The computational challenges raised by the high collision rates in the Large Hadron Collider are twofold. Firstly, it simply produces an extremely large amount of data to process. As a rough estimate, an LHC experiment with about 80 million

one-byte data channels (see Section 2.4) operating at the LHC’s 40 MHz crossing rate would produce about 3.2 PB/s of raw data. Although this rate is lowered significantly in practice due to, e.g. zero suppression, processing this volume of data remains challenging. Secondly, the large amount of data produced by LHC experiments cannot be stored as this would consume an infeasible amount of permanent storage space. As such, experiments must take split-second decisions about which data to keep and which data to discard. It is not uncommon for LHC experiments to retain data at a rate of approximately 1 kHz, which suggests a data retention rate of 0.0025 %. This process, often referred to as *triggering*, imparts weak real-time requirements on systems and requires low-latency computation.

Further complications arise from the large number of protons in the LHC. The LHC is designed such that, during a single bunch crossing, it is highly likely for there to be more than a single proton–proton collision, and this introduces combinatorial effects in the processing of the data which we explore in more depth in Chapter 4. The number of proton–proton interactions in a single crossing or event is usually referred to as the *pile-up* and is denoted μ . The average pile-up over a given period of time is usually written $\langle\mu\rangle$ and provides a useful metric for the computational power necessary to process each individual event [19, 45]. Pile-up has been growing rapidly in the ATLAS experiment, which was originally designed to operate at $\langle\mu\rangle \approx 23$: during the 2011–2012 period, average pile-up was $\langle\mu\rangle = 18.5$, during the 2015–2018 period $\langle\mu\rangle = 33.7$, and during the 2022–2023 period $\langle\mu\rangle = 46.5$. Considering the fact that the time to process data scales superlinearly with $\langle\mu\rangle$, this growth poses a real computational challenge.

This brings us to the primary motivation of this thesis. The Large Hadron Collider is expected to undergo an upgrade, referred to as the High-Luminosity Large Hadron Collider (HL-LHC) which is expected to significantly increase its luminosity. Through larger and tighter bunches, the HL-LHC upgrade project will cause a significant increase in pile-up and, thereby, pose an even greater computational challenge. In addition, LHC experiments will upgrade their detector systems, thereby also providing more data to process. The HL-LHC is expected to start operations in 2029, and so the clock is ticking to prepare software and hardware for metaphorical deluge of data that the HL-LHC will produce.

2.4 Particle Detectors and Experiments

A particle accelerator on its own has as much value to collider physics as a computer does to computational science: both are fundamentally necessary for their corres-

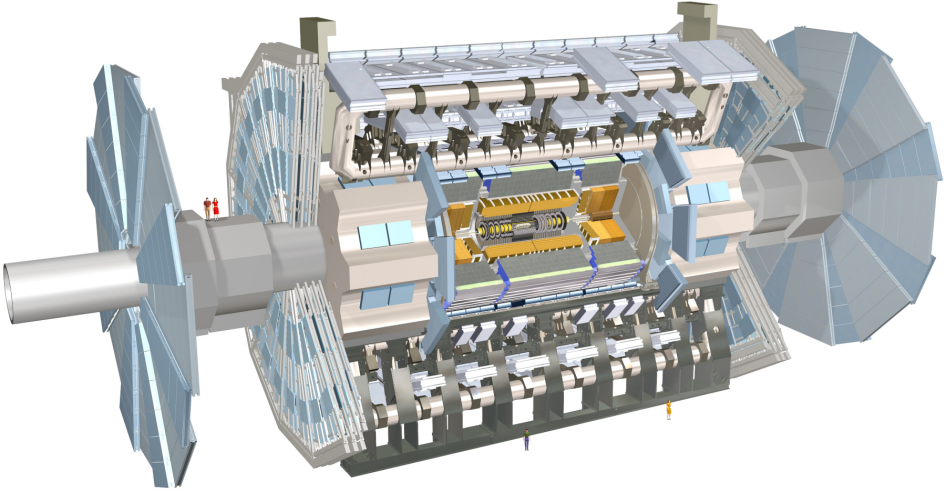


Figure 2.3: A cutaway diagram showing the ATLAS detector, one of the experiments on the Large Hadron Collider. Image provided by CERN [46].

ponding scientific processes, but they are tools that must be properly harnessed. A computer is of little use without software to exploit its computational prowess, and a particle accelerator requires detectors to record the complex physical processes that take place when collisions take place. These days, detectors are large constructs erected around locations where particles can be brought to collision. Examples of LHC experiments include ATLAS – a cylindrical experiment 46 m long and 25 m in diameter weighing 7 000 000 kg [47], shown in Figure 2.3 – and CMS – slightly smaller at 21 m long and 15 m in diameter but significantly heavier at 14 000 000 kg [48]. In this thesis, we will primarily discuss the ATLAS experiment, although our findings are broadly applicable to CMS and other experiments. The two beams – which are enclosed by the *beam pipe* – pass through the centre each of these experiments where they are made to cross, causing particle collisions in the so-called *interaction point* which can then be observed. These experiments then function as large three-dimensional cameras, observing the trajectories of the new particles that are produced in the collisions.

The heterogeneous nature of the particles that are created in collision experiments – see Section 2.1 – as well as the various properties that we wish to measure – including their trajectories and energies – demands a variety of detection techniques. Modern particle detectors – including both ATLAS and CMS – are commonly (but not necessarily) equipped with four distinct types of detection

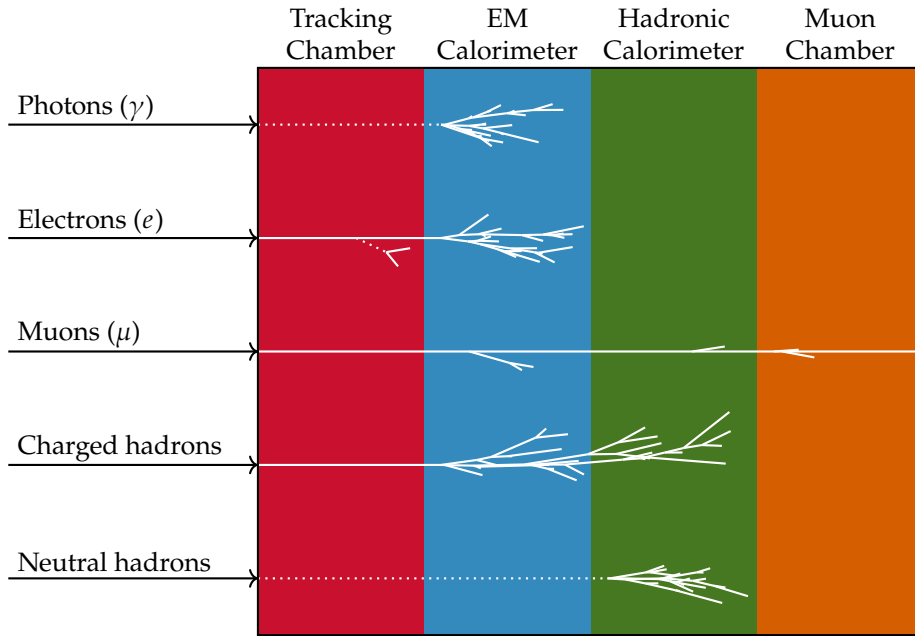


Figure 2.4: A simplified four-part particle detector consisting of – in order of increasing distance from the interaction point – a tracking chamber, an electromagnetic calorimeter, a hadronic calorimeter, and a muon chamber. The signatures of different particles is illustrated. The trajectories of neutral particles – shown as dotted lines – cannot be directly detected. Graphic adapted from Lippmann [49].

systems as shown in Figure 2.4. The so-called *tracking chamber* sits closest to the point of collision and is designed to record the trajectories of charged particles, but to let them pass through largely unimpeded. The electromagnetic and hadronic calorimeters are designed to stop and record the energy of particles that interact with the electromagnetic and strong nuclear forces, respectively. Finally, an additional tracking chamber is installed on the outer parts of the detector to track charged particles which pass through both calorimeters, principally muons.

The use of multiple detection systems is critical to the identification of particles due to the different *signatures* that particles exhibit. Using the aforementioned four-tier design we can distinguish five unique signatures, all shown in Figure 2.4. Photons are uncharged particle and, as such, do not interact with the tracking chamber; they can be detected only in the electromagnetic calorimeter. Electrons are charged and, therefore, are tracked in the inner tracking chamber and subsequently cause a shower in the electromagnetic calorimeter. Photons and electrons can thus

be distinguished by their respective absence and presence in the tracking chamber. Charged hadrons, like electrons, can be detected in the tracking chamber but will create showers that extend into the hadronic calorimeter. Neutral hadrons interact with neither the tracking chamber nor the electromagnetic calorimeter, except through hadronic processes such as nuclear capture. Finally, muons traverse through the tracking chambers and calorimeters unimpeded, and are therefore easily identified.

In this thesis, we focus on the development of software for the reconstruction of particle tracks in the innermost tracking chamber. The reconstruction of trajectories of calorimeter showers requires fundamentally different algorithms, which we consider outside of the scope of this thesis. Previous and ongoing efforts have worked towards the development of heterogeneous calorimetry software [50, 51]. The algorithms and implementations we develop in this thesis are applicable also to the muon chamber, although this is not our primary focus; muon chambers have different goals and operating parameters to inner tracking chambers, featuring significantly lower detector resolution and occupancy – i.e. the number of points at which particles are detected – but a greater ability to distinguish the momenta of particles due to the larger distances involved [52]. Due to the lower occupancy, track reconstruction in muon chambers may – depending on the design of the detector – be performed using Hough transform–based methods which are efficient – including on massively parallel architectures [53] – but struggle to handle dense environments like inner tracking chambers due to track multiplicity and scattering effects [54].

Many tracking chambers consist of so-called *sensitive surfaces* which can detect the passing of charged particles. Such surfaces are remarkably similar to those used in digital cameras, which is to say they consist of semiconductors; when a charged particle passes through such a surface, it creates an electron and an electron hole which can then be detected. Most – but not all – tracking chambers are equipped with an onion-like set of layers around the beam pipe; the ATLAS Inner Detector tracking chamber is visible in the centre of Figure 2.3. Note that this implies that, unlike photographic sensors which consist of a single plane, tracking chamber contain complex three-dimensional geometries of rotated detectors. As more layers are added to a tracking chamber, the circumference of these layers increases and it becomes prohibitively expensive to tile them with detectors. For this reason, most tracking chambers use cheaper and less precise detectors as the distance to the beam pipe increases.

In the ATLAS experiment, the pixels closest to the beam pipe have a size of

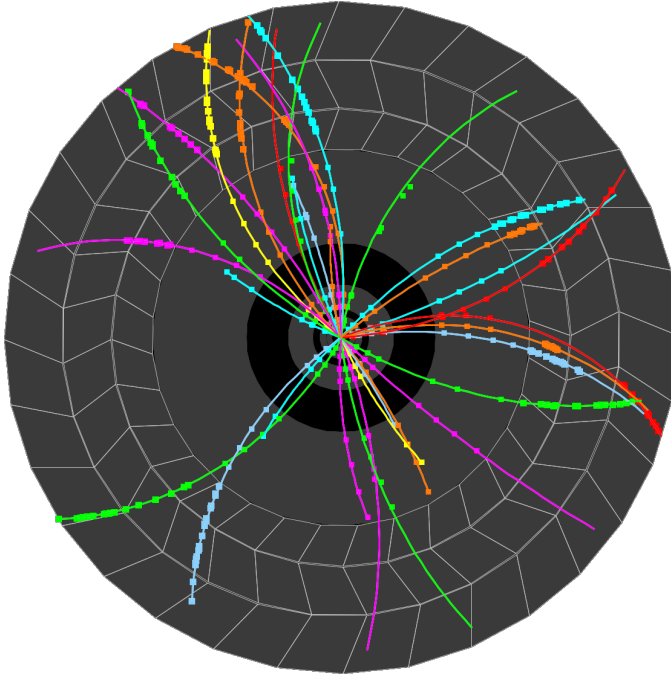


Figure 2.5: An example of a single collision event in the ATLAS experiment. In this image, the beam pipe runs perpendicular to the page. Shown are the discrete measurements left in the detector surfaces (squares) and the reconstructed trajectories (lines). Note that this is a two-dimensional projection of a three-dimensional event. Image adapted from CERN [55].

$50\text{ }\mu\text{m} \times 250\text{ }\mu\text{m}$ [56], and the outer layer pixels have a size of $50\text{ }\mu\text{m} \times 400\text{ }\mu\text{m}$ [57]⁶. In total, ATLAS has four pixel layers with a total of 1736 sensor modules, as well as the so-called end-caps (perpendicular to the beam pipe) with a total of 288 sensor modules. This gives ATLAS a total of 92 million individual pixels. Outside the pixel layers, ATLAS also has four layers of so-called Semiconductor Trackers, which are significantly less precise compared to the pixel detectors [47].

It is through the use of silicon-based sensitive surfaces that we acquire knowledge of where particles have traversed the experiment. Note, however, that these surfaces do not track the continuous trajectories of the particles: they record only at discrete locations. Furthermore, particles leave no identifying information in these detectors, so there is no inherent information about which measurements belong to the same particle. It is this combination of factors – the lack of continuous

⁶For comparison, a modern full-frame photo camera has pixels of around $6\text{ }\mu\text{m} \times 6\text{ }\mu\text{m}$.

measurements and the lack of identifying information – that gives rise to the track reconstruction problem: which measurements belong to the same particle, and what is the trajectory of that particle through the experiment? An example of the solution to a track reconstruction problem is shown in Figure 2.5.

As seen in Figure 2.5, the tracks of particles approximate arcs or – in three dimensions – helices. Note that perfect arcs are only formed in ideal circumstances with constant magnetic fields, without interactions with the detector material, and without energy loss; tracks observed in the real world are not perfectly helical. The reason for this is that most tracking chambers are permeated by a powerful magnetic field which curves the trajectories of the particles according to the Lorentz force; given a vector-valued function representing the magnetic field at a position \vec{x} , $\vec{B}(\vec{x})$, an electric field $\vec{E}(\vec{x})$, as well as the momentum and charge of a particle at \vec{x} , denoted \vec{p} and q respectively, we find the force acting on that particle according to Equation 2.6:

$$\vec{F}(\vec{x}, \vec{p}, q; \vec{E}, \vec{B}) = q(\vec{E}(\vec{x}) + \vec{p} \times \vec{B}(\vec{x})) \quad (2.6)$$

In particle physics, experiments are usually defined such that the electric field \vec{E} is zero and the magnetic field \vec{B} is roughly constant, uniformly aligned – often along the beam pipe – and quite powerful. The intuition is that the force enacted on the particle is always perpendicular to its momentum, causing it to curve but not decelerate. In addition, the curvature of the particle depends on its energy; just like how a car going 120 km/h has a larger turning radius than a car going at a walking pace, a particle with more energy will curve less strongly than a particle with lower energy. It is here that we find one of the primary motivations for track reconstruction: knowing the radius of a particle’s curvature allows us to estimate its energy, which is crucial to deciphering which processes took place in a collision [23]. Another reason why it is useful to know the trajectory of a particle is so that the track can be extended into, e.g. the calorimeters, allowing us to associate measurements in different parts of the detector with tracks identified in the inner tracking chamber.

In order to accurately reconstruct tracks in an experiment, the geometry of that experiment must be known, i.e. the location of each of the thousands of detector surfaces must be precisely recorded, as must the location of any non-sensitive materials, e.g. support structures which can influence the trajectories of particles. It is an open question how detector geometries can best be stored for different processor architectures, and there are several ongoing developments including

but not limited to GEOMODEL [58], DD4HEP [59], and DETRAY, the latter of which is designed specifically for GPGPU architectures [60].

Track reconstruction usually takes place in one of two scenarios. The first scenario is the so-called *online* or *trigger* scenario in which track reconstruction is used to decide whether to store or discard data as it is recorded from the detector. Online reconstruction usually happens in a two-step process: the first step, implemented on specialised hardware, makes a split-second decision based on very rudimentary reconstruction in order to reduce the data volume. In ATLAS, this so-called *low-level trigger* reduces the data rate from 40 MHz to about 100 kHz. A second step, the so-called *high-level trigger* is implemented on commodity hardware and further reduces the data rate to about 1 kHz. Online environments pose soft real-time requirements on track reconstruction software and requires low latency. The second scenario is the so-called *offline* scenario in which data is processed after it has been stored, which requires high throughput. This thesis is concerned with both offline processing and high-level triggering, as this is where the ATLAS experiment performs track reconstruction.

3

Parallel and Massively Parallel Computing

'Making processors faster is increasingly difficult,' John thought, 'but maybe people won't notice if I give them more processors.'

— **James Mickens**
(Professor of computer science)

Parallelism is the idea that two or more individuals – whether they are animals, humans, or computer systems – can simultaneously and cooperatively work towards a single goal in an efficient way. It is an idea that is both trivially simple to grasp and dauntingly difficult to do well. It can take many shapes and the development of parallelism has been one of the most critical drivers in modern high-performance computing. In fact, the development of parallelism was an inevitability imposed on computing by none less than the fundamental laws of physics. Parallelism has been around since the very start of computing, yet it is arguably closer to its inception than it is to being a fully understood idea. This thesis serves to advance our understanding of the principles and practice of parallel computing, and this section serves as a brief introduction to some of the core ideas of parallelism.

3.1 History of Hardware Performance

There are few concepts in computer science more tortured than Moore's law. For many years, academic publishing has been a veritable battleground for claims that the late Gordon Moore's eponymous observation is metaphorically alive, dead, or – like Schrödinger's famous cat – anything in between. Countless articles and theses have relied on Moore's law to motivate all kinds of ideas in computing, and some have gone on to wrestle the idea into fields well beyond semiconductor design.

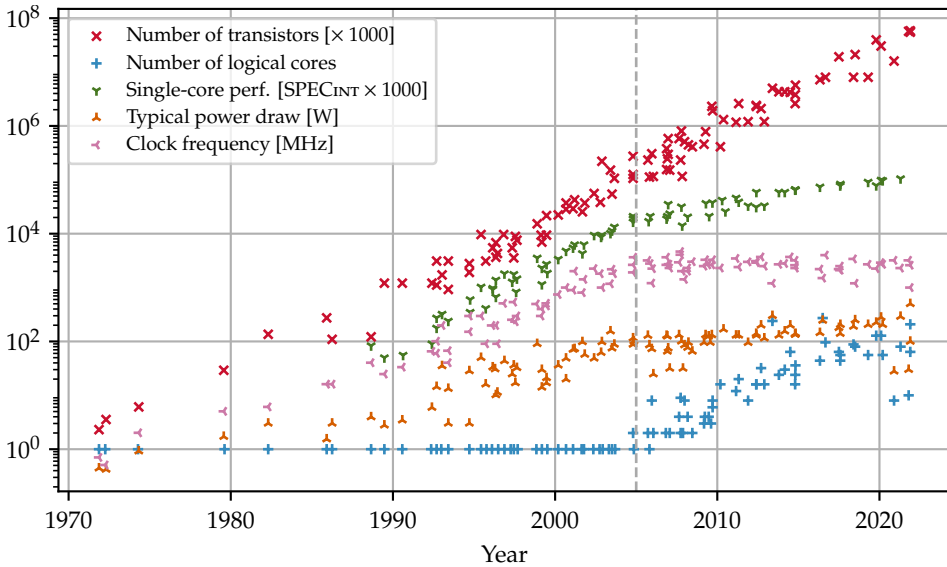


Figure 3.1: Trends in core count, transistor count, clock frequency, single-core performance, and power consumption in microprocessors over the last few decades. This plot is generated from data made available by Rupp [63] under the Creative Commons Attribution 4.0 International license.

The fact of the matter is, however, that Moore’s law is such an eminent notion that we too have succumbed to its uncanny ability to characterise the last sixty years of microarchitecture development: despite our best efforts, we can recognise the cliché of starting a chapter with a description of Moore’s law but we cannot avoid it.

Briefly, Moore’s law describes the idea that the number of transistors in microprocessors doubles roughly every two years [61]. What makes Moore’s idea particularly prescient is that it was posited in 1975¹ when the hottest news in the microprocessor world was the Intel 8080 with its 4500 transistors [62], and that it has held with remarkable precision well into the twenty-first century – a time in which processors with more than one hundred billion transistors are available on the consumer market. The uncanny adherence of the number of transistors in microprocessors to the exponential growth predicted by Moore is illustrated by Figure 3.1.

¹It is worth mentioning that Moore’s law is a decade older than this; the original 1965 statement was too optimistic, however, predicting a doubling every single year rather than every two years.

Moore's law is enabled principally by the ability of semiconductor manufacturers to shrink the size of individual transistors and to manufacture them more densely. Indeed, if transistors had stayed the same size they were when the aforementioned Intel 8080 came out, a modern high-end processor would have to be about the same size as a tennis court². The use of such a device would, obviously, be rather impractical even after the installation of the extraordinarily large cooling apparatus that would be required to keep it from immediately destroying itself through the production of waste heat. The decrease in transistor sizes has been both a remarkable feat of engineering as well as one of the prime reasons that modern-day computers are as powerful as they are. As Figure 3.1 shows, transistor counts are still growing at roughly the same rates that they were before. However, there are physical limits on the size of transistors [65], and it remains to be seen to which extent transistor technology can still be shrunk.

Although Moore's law predicts transistor counts with a high degree of accuracy, it does not *directly* predict the performance of microprocessors, similar to how the number of bricks in a house does not determine how comfortable it is to live in. Very roughly speaking, the transistors in a processor can be used either to perform computations directly, e.g. additions and multiplications, or to improve the quality of the schedulers, register renaming schemes, instruction queues, and so forth, i.e. the meta-computation which allows the processor to more efficiently perform the task presented by the programmer. Expanding the meta-computational capabilities of processor cores faces diminishing returns, however. As an example, *out-of-order execution* provided significant performance benefits when it was first introduced in the CDC 6600 [66], but devoting more transistors to it does not improve performance much beyond certain points. Out-of-order execution is the ability of a processor to execute instructions in a different order than they are issued according to the source code that is executed, and this is generally accomplished using a fixed-sized instruction window which determines how many instructions can be reordered. Increasing the instruction window size has been shown to exhibit diminishing returns with respect to the performance gained [67], and to greatly increase both the complexity and power consumption of the processor [68].

Another common paradigm to improve core performance is to add hardware implementations for increasingly complex computations. This is demonstrated well by so-called *fused multiply-add* instructions, which can compute $a \cdot b + c$ with

²The 1974 Intel 8080 had 4500 transistors on a 20 mm² die [62]; an NVIDIA A100 GPU features about 54.2 billion transistors [64] which would – at the transistor density of the Intel 8080 – give a die area of 240.9 m²

the same latency and throughput as either an addition or a multiplication. In other words, it allows us to perform twice as many operations and thus increases performance. Since additions and multiplications are such common operation, the fused multiply-add operation has the potential to provide significant performance benefits. The problem with this approach is that it doesn't scale indefinitely, e.g. it is doubtful that many applications would benefit from, say, a 'fused add-then-multiply-then-square-then-divide' operation; such an operation would be so niche that it would benefit only a very small class of problems. As an example of how specific some instructions in modern processor are, the GFNI instruction set extension to x86 introduces the VGF2P8AFFINEINVQB instruction, which computes an affine transformation in the finite field \mathbb{F}_{2^8} with respect to the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. Although VGF2P8AFFINEINVQB is very useful for AES cryptography [69], it is unlikely to accelerate a broad range of applications.

The fact that we cannot keep adding more and more advanced features into processor microarchitectures would not be a big problem if we would be able to keep increasing the frequencies at which processors operate. Unfortunately, as Figure 3.1 shows, frequency scaling suddenly stagnated around the year 2005. Before 2005, the increase in processor clock frequencies was broadly described by Dennard scaling [70], which – among other things – states that on a generation-by-generation basis: (1) the size of individual transistors shrinks by about $\kappa = 30\%$; so (2) the area of a die shrinks by $1 - (1 - \kappa)^2 = 51\%$; so (3) the capacitance of the circuit C decreases as the area of the die over the distance, i.e. by $1 - \frac{1-51\%}{1-\kappa} = \kappa$; and (4) the voltage and current of the circuit V and I both decrease by κ as it scales with distance; so (5) the transistor delay time $t = \frac{VC}{I}$ decreases by a factor κ ; so (6) the frequency $f = \frac{1}{t}$ increases by approximately $\frac{1}{1-\kappa} = 42.8\%$. In simpler terms, there was a motivation – grounded in the laws of physics – for why processor frequencies could be scaled year-over-year. When transistor sizes reached about 65 nm, however, voltage leakage started to dominate the circuit voltage, meaning that the latter could no longer be decreased and, as a result, delay times could no longer be reduced such that clock frequencies had to remain the same [71].

3.2 Parallel Computing

As we have seen, our ability to grow the performance of single processor cores has become severely hampered by our inability to increase the clock frequencies, as well as by diminishing returns on performance gains due to increased core complexity. These effects are apparent in Figure 3.1 where – around the year 2005 – the

trend for processor clock frequencies suddenly reaches a stable plateau and where the single-core performance of processors starts to grow significantly less quickly. Thankfully, hope is not lost: in spite of the aforementioned effects, processor manufacturers have managed to continue to deliver remarkable performance benefits in their processors, and the primary driver of this phenomenon is *parallelism*. Parallelism comes in various forms, and we will briefly touch upon so-called data-level parallelism, instruction-level parallelism, and task-level parallelism, as these are arguably the main drivers of parallel programming. Furthermore, it is crucial to understand these forms of parallelism in order to understand the architecture of graphics processing units.

3.2.1 Instruction-Level Parallelism

Instruction-level parallelism, often referred to as ILP, is the idea that parallelism can be achieved at a very low level by running instructions, or even units of computation smaller than single instruction, in parallel inside of a single core. ILP comes in various forms and is one of the primary drivers of the continued growth of single-core performance that we still see today. It is a common misconception that parallelism only happens between cores (see Section 3.2.3 on task-level parallelism), but indeed there is also a significant amount of parallelism inside of each core. In this section, we will discuss four types of ILP, namely pipelining, superscalar execution, out-of-order execution and speculative execution.

Pipelining describes the idea that the execution of individual instructions follows a series of inherently sequential steps, and that the hardware that implements these steps can be efficiently reused in order to increase performance. We will briefly consider the canonical five-stage RISC pipeline, but it is worth noting that the pipelines in many modern processors are significantly more complex. In the canonical pipeline, executing an instruction involves five stages: instruction fetching, instruction decoding, execution, accessing memory, and writing back to the register file. In a non-pipelined processor, an instruction would have to pass all five stages before the next instruction could be issued, but this would leave four out of five pipeline stages idle at any given time. Using pipelining, new instructions can be issued more rapidly as the hardware used for each of the stages is freed up [72]. Instruction pipelines are vulnerable to a variety of so-called *hazards* [73] which threaten instruction throughput, but the description of such hazards as well as methods to mitigate them is beyond the scope of this thesis. It is worth noting that the length of the longest pipeline stage also determines the maximum clock

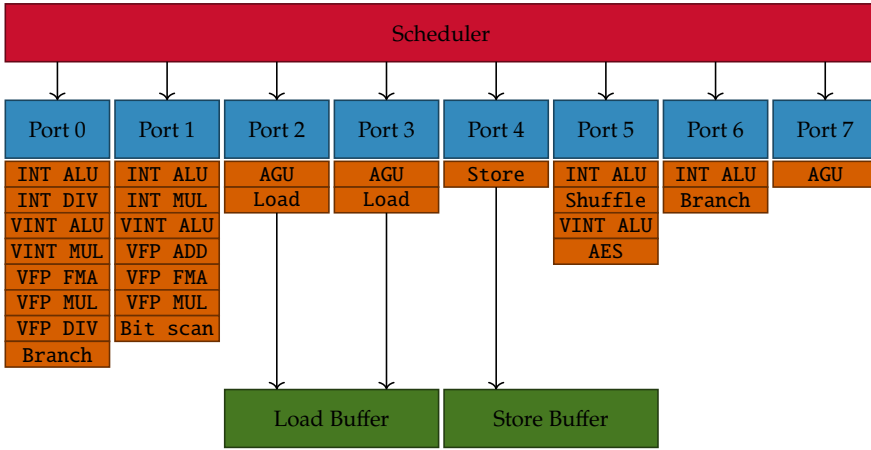


Figure 3.2: A schematic overview of the execution units and ports of the Intel Haswell core. The abbreviation INT denotes integers, VINT denotes integer vectors, VFP denotes floating point vectors, ALU denotes arithmetic logic units, DIV denotes dividers, MUL denotes multipliers, FMA denotes fused multiplier-adders, and AGU denotes address generation units. Figure adapted from Hofmann, Fey, Eitzinger, Hager and Wellein [74].

frequency of a processor, as the processor must guarantee that every pipeline stage can complete in a single cycle.

Another form of ILP, often implemented side-by-side with pipelining, is so-called *superscalar execution*. The term superscalar execution implies computation on more than one scalar at a time, but is – perhaps confusingly – strictly different from vectorisation, which is a form of data-level parallelism (see Section 3.2.2). Rather, superscalar execution describes the fact that many modern processors have multiple so-called *execution units* which can execute similar instructions [75]. As an example, Figure 3.2 shows the back-end of the Intel Haswell microarchitecture. Critically, the Haswell microarchitecture contains several execution ports which can each execute a fixed set of instructions. For example, port 2 can serve as either an address generation unit or a load unit but, more importantly, loads can be executed by either port 2 or port 3. In other words, the Haswell microarchitecture can execute two loads at the same time [74] – or more if we consider pipelining. Similarly, it allows us to perform multiple integer or floating point arithmetic operations at the same time. Finally, superscalar execution allows us to benefit from the ability to execute both floating point and integer operations at the same time: a prime example of this is in tight loops over floating-point values, where

the floating point execution units can perform the necessary computation while the integer units can – at the same time – increment and compare the loop counter.

The final two ILP-related concepts that we discuss in this section are speculative execution and out-of-order execution, which we discussed earlier in Section 3.1. These techniques differ slightly from pipelining and superscalar execution, because they are not ways to *enable* ILP, but rather ways to *enhance* existing ILP. Out-of-order execution, for example, allows the processor to fill unused pipeline stages or execution units by executing instructions from further ahead in the program. Similarly, speculative execution allows a processor to start executing instructions following a conditional branch in a speculative way, i.e. without any guarantees that that branch will be taken. Like out-of-order execution, the primary goal of speculative execution is to provide additional operations which can be pipelined and executed in a superscalar fashion [76]. Ideally, superscalar execution provides no downsides if a branch is not taken, and provides significant upsides if the branch prediction is correct.

3.2.2 Data-Level Parallelism

Data-level parallelism, sometimes referred to as DLP, refers to the idea that – in a very large number of applications – the goal of the programmer is to perform an operation on not one but on a large number of data points, and that those operations can be performed in parallel. Consider, for example, the addition of two four-element floating point vectors; this operation would traditionally take at least four instructions which each have to go through the entire instruction pipeline³. Modern processors are equipped with the ability to perform these four additions in a single instruction through the so-called *single-instruction multiple-data* paradigm as described by Flynn [77]. This form of data-level parallelism is also frequently described as *vectorisation*, as it concerns computation on vector data. In commodity hardware, support for data-level parallelism generally comes in the form of instruction set extensions. For x86, this began with the introduction of the MMX instruction set extension in 1997 which allowed 64-bit integer registers to act either as such, or as vectors of two 32-bit integers, four 16-bit integers, or eight 8-bit integers [78]. In 1999, x86 was further expanded with the Streaming SIMD eXtensions (SSE) which introduced 128-bit registers which could, interestingly, *not* be used to store 128-bit numbers but rather served as vectors of either two 64-bit double-precision floating point numbers or four 32-bit floating point numbers [79].

³Not to mention that these four instructions have more subtle effects, like potentially reducing the locality in the instruction cache.

Importantly, SSE allows processors to execute an operation on an entire vector in the same time that it would take to perform that operation on a single scalar. SSE would go on to replace x87 as the de facto floating point format for x86 [80]. Advanced Vector Extensions (AVX) added support for 256-bit vectors in 2008 and this was extended to 512-bit in 2015 in the form of AVX-512. In the ARM world, the Scalable Vector Extensions (SVE) adds support for vectors up to 2048 bits in length [81].

It is trivial to see that DLP is a big driver of single-core performance through parallelism; replacing operations on scalars by operations on, e.g. vectors of 16 elements (as is the case for 512-bit vectors of 32-bit floating point numbers) effects a speed-up of 16 times⁴. However, obtaining such speed-ups is not always easy. Unlike instruction-level parallelism which – in modern computing – is enacted entire in the processor core with little help from the compiler and virtually no help from the programmer directly, data-level parallelism requires much more intervention. Indeed, vector instructions must be provided to the processor by the compiler, and this is no trivial feat: the automated vectorisation of code is an active field of research [82] and compilers often fail to vectorise all but the simplest loops, partially due to the rigorous constraints on, e.g. data alignment. It is often the responsibility of the programmer to ensure that a program can be executed in a data-parallel fashion through the use of so-called *intrinsics* – function which compile directly to a specific instruction – or through the use of libraries that support vectorisation such as EVE [83, 84, 85]. Higher-level libraries such as those for linear algebra also commonly support data-level parallelism, including EIGEN3. Finally, so-called array programming has also become more popular as it maps naturally onto the data-parallel capabilities of modern processors; the popular NUMPY Python library is an example of an embedded array programming language [86], and dedicated array programming languages include APL [87], JULIA [88], and FUTHARK [89]. Using such (embedded) languages can bring higher performance as it forces programmers to write programs in ways that can be more easily and naturally compiled to data-parallel code.

3.2.3 Task-Level Parallelism

The final form of parallelism that we discuss in this chapter is task-level parallelism, often referred to as TLP. Compared to ILP and DLP, TLP is a far more

⁴Throughout this thesis, we use speed-up S to denote how many times *slower* a previous version of a program is compared to the current one, i.e. $S = T_{\text{old}}/T_{\text{new}}$.

coarse-grained form of parallelism in which programmer-defined tasks are distributed over multiple threads or machines. Because task-level parallelism features multiple independent instruction streams, it allows for the execution of different kinds of operations in parallel; e.g. it is task-level parallelism that enables us to open a browser and a document reader on our laptops at the same time and it can be used to rapidly process large amounts of data. Where ILP and DLP are forms of parallelism inside a single core, TLP is a form of parallelism which exists principally between cores, or even between entirely different computers. Common forms of task-level parallelism are multi-threaded and multi-process computing. In multi-threaded computing, multiple instruction streams operate on and manipulate shared memory, which allows easy communication between threads. In multi-process computing, processes with separate virtual memories operate simultaneously; although multi-process computing requires additional communication between processes – which can be expensive – it is much more easily scalable to distributed systems.

An intuitive but powerful idea in parallelism which is particularly relevant to task-level parallelism – although it also affects data-level parallelism – is *Amdahl's law*. Although Amdahl posited it much more informally in his original paper, Amdahl's law states that the speed-up S depends on the fraction p of a program that is improved, and the amount by which that part is sped up s as in Equation 3.1 [90]:

$$S = \frac{1}{(1 - p) + \frac{p}{s}} \quad (3.1)$$

In parallel computing, it is often assumed that the part of the program which is sped up is sped up using parallelism, and that it is sped up by a factor that is equal to the number of processors. It is also common to use Amdahl's law to place an upper limit on the speed-up that can be achieved through parallelism regardless of the number of processors available, as in Equation 3.2:

$$\lim_{s \rightarrow \infty} S = \frac{1}{1 - p} \quad (3.2)$$

Amdahl's law shows us that the use of parallelism to speed up computer programs is very sensitive to sequential parts of the code – i.e. all non-parallel parts, a fraction $1 - p$; a program which runs sequentially for only 5 % of its runtime is limited to a speed-up by a factor twenty, no matter if we use a dozen, a thousand, or a million processors to run it. Although Amdahl's law leads to rather pessimistic predictions, it is based on the assumption that the application scales *strongly*, i.e.

that the total problem size remains the same. This is referred to as *strong scaling* and it is not always representative of what happens in the real world.

In reality, it is common for problem sizes to scale with the number of processors – or, equivalently, it is common for the number of processors to scale with the problem size – which is referred to as *weak scaling*. Where strong scaling is governed by Amdahl’s law, weak scaling is governed by *Gustafson’s law* which provides a more optimistic view of parallel computing. Under Gustafson’s law, given in Equation 3.3, the parallel fraction of the program p grows with the number of processors s , while the sequential part $1 - p$ remains the same [91]:

$$S = (1 - p) + ps \quad (3.3)$$

It is worth noting that the categories of parallelism listed in this section are somewhat flexible, and it is not always clear where a given technique belongs. For example, Simultaneous Multiprocessing (SMT) – referred to as *Hyper-Threading* in Intel parlance – refers to the ability of many modern cores to execute two or more instruction streams or threads on a single core. From a programmer’s perspective, this is task-level parallelism, but from the perspective of the core, SMT relies on the assumption that these threads can use execution units that the other threads are not using. It is, therefore, also a form of instruction-level parallelism. Interpreted differently, it is task-level parallelism enabled by instruction-level parallelism, or even instruction-level parallelism masquerading as task-level parallelism. Although the classification of SMT is not particularly important, the insight that the borders between these flavours of parallelism are quite blurred will be important for us to understand how GPGPU programming works.

3.3 Graphics Processing Units

As computers became more commonly available in the sixties and seventies of the twentieth century, video games became increasingly popular. At first, games were mostly text-based; the computer would present the player with textual information and a prompt, upon which the player would input some text-based response. Over time, graphical interfaces to video games were developed which gave players a more immersive experience. The rendering of computer graphics was, however, a daunting task: rapidly changing images had to be rendered to a screen – or, more technically, a frame buffer – many times per second.

The 1978 arcade game *Space Invaders* – see Figure 3.3 – is perhaps one of the

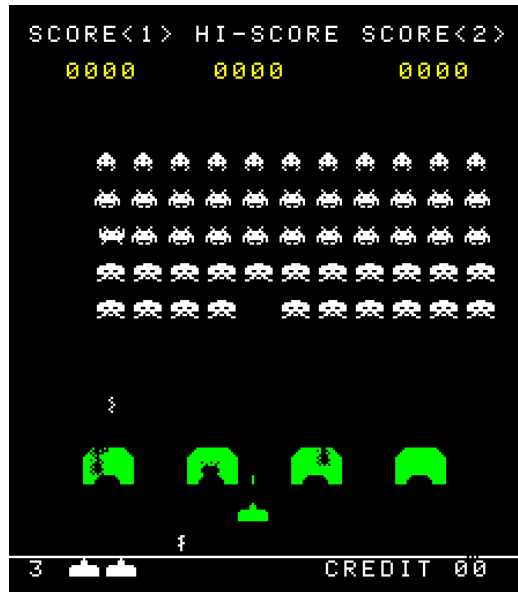


Figure 3.3: A screenshot of the 1978 video game *Space Invaders*, developed at Taito Corporation by Tomohiro Nishikado, emulated using PCs [92].

most iconic video games ever developed [93]. In *Space Invaders*, the player pilots a space ship at the bottom of the screen and engages in a space battle with a fleet of eponymous invaders from space. One of the challenges in *Space Invaders* lies in the fact that the space ships which the player is attempting to shoot are constantly moving horizontally across the screen. The original arcade version of the game ran on a 256×224 display for a total of 57 344 pixels. The act of moving this many pixels left and right multiple times per second was a computationally intensive task, and it proved too much to handle for the Intel 8080 microprocessor which powered the arcade machine. In order to allow the *Space Invaders* game to run smoothly, the Intel 8080 microprocessor was accompanied by multiple barrel shifting circuits which served solely to move pixels across the screen; in other words, it was circuitry designed to accelerate computer graphics.

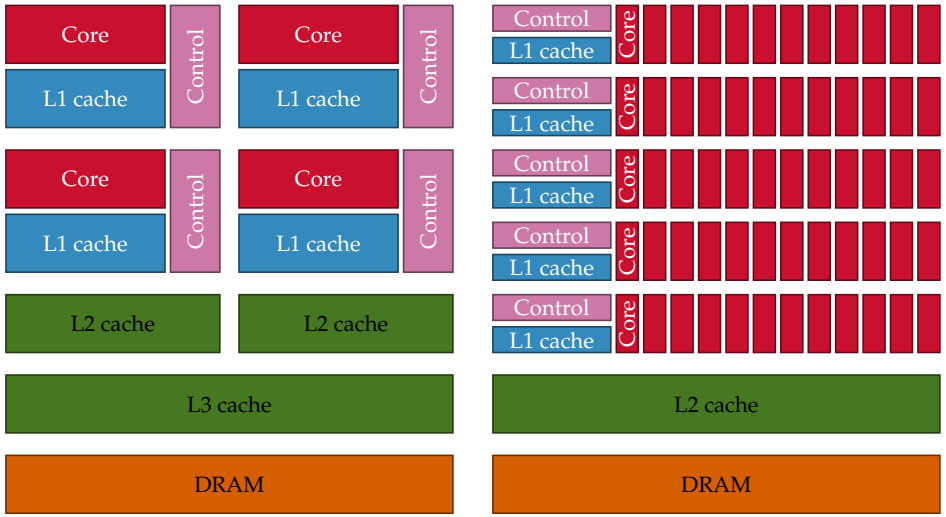
Although *Space Invaders* was by no means the first application to feature accelerator circuits for the purpose of rendering graphics – the 1975 arcade game *Gun Fight*, also developed by Tomohiro Nishikado, featured a very similar array of bit-shifting circuits – it is a poignant example of a hardware design pattern which later lead to the development of the Graphics Processing Unit (GPU). In order to facilitate the design of arcade games, Fujitsu developed the MB14241,

an Application-Specific Integrated Circuit (ASIC) which encapsulated the barrel shifting required for various games. In the decades that followed, GPUs developed into vastly more complicated and computationally powerful components. This, in turn, led to significant advances in many fields of computational science as well as in artificial intelligence. The remainder of this section serves to describe the design of GPUs in general terms, as well as the usage of GPUs for general computing. We also briefly discuss the degree to which CPU and GPU architectures have re-converged in recent years.

3.3.1 GPU Architecture

The MB14241 is – in some sense – a quintessential example of GPU architecture; given that such a large number of pixels had to be moved around in a fixed pattern to make *Space Invaders* playable, it made little sense to use a general-purpose processor like the Intel 8080. The Intel 8080 was equipped with the ability to take conditional branches, to move data to arbitrary locations and to perform a broad range of operations, but all of this computational prowess wasn't necessary to display the metaphorical Salsa dance of the spaceships in *Space Invaders*. Rather, *Space Invaders* demanded the ability to perform the exact same operation many times in a single second, and this is not necessarily what the Intel 8080 was designed to do. The same problem exists in many modern computer graphics applications: similar operations have to be performed for millions of pixels at a time, dozens of times per second. This is a form of data-level parallelism, with the subtle nuance that the parallelism does not necessarily stem from the input data, but rather from the structure of the output data: the pixels on the screen. In the world of computer graphics, computations which are performed in parallel are referred to as *shaders*, which follows from the fact that they are used to compute the shade – i.e. the colour, brightness, transparency, etc. – of what is on the screen. Besides pixel-by-pixel computations, the term shader can also be used to describe, for example, computations on objects in a three-dimensional scene [94]; regardless, the common theme is that shaders are often run in parallel.

The architecture of GPUs reflects the parallel nature of shader computation. In fact, the degree to which shaders are computed in parallel is so great that it is often far greater than the parallelism that is found in applications for CPU-like architectures, which is why we refer to the parallelism in GPUs is often referred to as *massive* parallelism. At the time of writing, a top-end commodity x86-64 CPU may have around 128 cores and – through simultaneous multithreading – 256



(a) In a MIMD architecture, each core has its own control and L1 cache. (b) In a SIMT architecture, many cores share the same control and L1 cache.

Figure 3.4: A schematic representation of the difference between MIMD and SIMT architectures, inspired by the NVIDIA CUDA C++ Programmer’s Guide [97]. Note that this image draws a potentially unfair comparison between the CPU cores and GPU cores, which we discuss in Section 3.3.3.

threads [95]. A top-end GPU, however, has can simultaneously host up to 270 336 threads⁵ [96]. Although it is far from fair to directly compare the CPU threads to GPU threads in terms of performance – as we will discuss in Section 3.3.3 – it does motivate the fact that programming GPUs have different architectures and that they need to be programmed in different ways.

The computational demands placed on GPUs include massively parallel execution of relatively rigid shaders. Because these workloads are relatively predictable and because all threads are executing roughly the same code, GPUs have – compared to CPU architectures – relatively high raw computational requirements and relatively little meta-computational requirements, e.g. they do not benefit much from branch prediction, speculative execution, out-of-order execution, and so forth. In CPU-based architectures, significant numbers of transistors are dedicated to this meta-computation, which is often referred to simply as the *control* of the core. In GPU architectures, the control is often simplified significantly and, even more

⁵This figure is given by the fact that the NVIDIA H100 GPU has 132 Streaming Multiprocessors (SMs), each of which can host up to 2048 threads.

importantly, it is shared between a large number of threads. The sharing of control means that cores are bound to execute the same instruction stream, which gives rise to the so-called Single Instruction Multiple Threads (SIMT) compute model. An illustration of the difference between a CPU-like MIMD architecture and a GPU-like SIMT architecture is shown in Figure 3.4. Note that, in addition to the aforementioned SIMT thread model, GPUs usually have simplified cache hierarchies and high memory bandwidths, as workloads in computer graphics are often memory-intensive.

It is also worth noting that the origin of GPUs in graphics processing also dictate the precision of computation that can be performed on them. Most commercially available GPU architectures feature primarily 32-bit IEEE 754 floating point ALUs and very few 64-bit ALUs. An NVIDIA GA102 *Ampere* GPU, for example, has a only 168 64-bit ALUs while it comes equipped with 10 752 32-bit ALUs; in other words, it has a 1 : 64 64-bit to 32-bit performance ratio [98]. This means that most GPUs are poorly equipped for applications that demand very high floating point precision. Exceptions to this rule of thumb include data centre GPUs such as the NVIDIA A100 and the AMD Instinct MI250X which feature 1 : 2 and 1 : 1 ratios [64, 99]. Recent developments in GPU architectures have further pushed for lower-precision computation; a primary example of this are NVIDIA *tensor cores* which support 19-bit, 16-bit, 8-bit, 6-bit, and even 4-bit floating point numbers at extremely high throughputs [64]. These low-precision compute units are designed primarily to support workloads in the field of artificial intelligence, which are an important design consideration in modern GPU designs.

3.3.2 General-Purpose GPU Computing

Early GPUs featured fixed-function shaders which performed a specific operation. In 2001, NVIDIA released the GeForce 3 series – codename NV20 – based on the Kelvin microarchitecture. The GeForce 3 series aimed to meet the demands of contemporary video games which aimed to set themselves apart through new graphical effects. Of course, it would have been infeasible for NVIDIA to design custom fixed-function hardware for every conceivable graphical effect that game developers might have wanted to incorporate into their games. Instead, NVIDIA incorporated programmable shaders into the GeForce 3 GPUs [100], which could be used to run programmer-defined shaders and afforded graphics programmers broad freedom to implement whatever effects they wanted. Programmable shaders greatly advanced computer graphics, but also gave rise to an even more important

development: that of general-purpose computing on GPUs.

After 2001, programmable shaders became increasingly advanced and independent of the remainder of the graphics pipeline. Simultaneously, computer scientists began to realise that GPUs could be used for general-purpose computation [101] – i.e. computations which are not necessarily related to computer graphics – which gave rise to the General-Purpose Graphics Processing Unit (GPGPU) paradigm. In 2008, NVIDIA released the Tesla microarchitecture – marketed as a ‘unified graphics and computing architecture’ – and, along with it, the *CUDA* programming model [102]. *CUDA* allows programmers to develop arbitrary computations in the C programming language which can then be compiled and ran on NVIDIA Tesla GPUs (as well as on later architectures). *CUDA* greatly increased the ease of programming in the GPGPU paradigm, and *CUDA* remains among the dominant programming models for massively parallel GPU architectures to this day.

The GPGPU paradigm leverages the relatively large amount of raw computational power inherent in the design of GPUs, but it also suffers from some downsides due to the simplified control that GPU cores have. The core downside of GPGPU programming is that groups of threads share a single instruction stream. In other words, instructions are decoded and issued to a group of threads – usually 16, 32, or 64 threads; this is referred to as a *warp* in NVIDIA parlance, although they are more generally referred to as *thread groups* – at the same time, rather than each thread having its own instruction stream. This is often referred to as *lockstep* execution, since each thread executes the same code at the same time. Of course, true lockstep execution is undesirable, as general-purpose programs can feature branching, and executing code which is not in the taken branch would violate the requirements of the program under execution. For this reason, GPGPU architectures usually feature conditional execution of some kind, in which a given thread can – based on some predicate set when a branch is encountered – decide whether to execute an instruction or not. Naturally, threads idling in order to avoid executing irrelevant instruction is a performance antipattern, and this so-called *thread divergence* is a common performance problem in massively parallel programs [103] and indeed, as we will see throughout this thesis, thread divergence is an important consideration in massively parallel track reconstruction.

The decreased complexity of the control in GPUs means that they are at a higher risk of performance degradation due to waiting. In CPU-like architectures, latency incurred due to pipeline stalls, cache misses, or other performance problems can readily be hidden with instruction-level parallelism enabled by out-of-order execution or speculative execution. Thankfully, GPU architectures commonly

employ other strategies to hide latency which are specifically enabled by their design; primarily, this comes in the form of zero-cost context switching. In CPU architectures, hardware threads may be interrupted by a variety of effects, including the scheduling and preemption of threads by the operating system. As such, it must be possible for a CPU architecture to store the complete state of a given thread – including the registers, the virtual memory space, and the instruction counter – so that execution can be resumed later. Because CPUs have such complex control which often relies on predicting and rescheduling future instructions, so-called context switching is expensive. Furthermore, the complexity of each individual instruction stream is so great that most x86-64 processors only permit two streams per core through simultaneous multithreading.

GPU architectures, in contrast, are not at risk of being context switched as their execution is not governed by an operating system in the way that a CPU is. Furthermore, GPU threads have much less complex states – in part because they do not have their own instruction stream – and this allows GPUs to hide latency through an ingenious implementation of instruction-level parallelism. In many GPU architectures, there are many more thread *slots* than there are execution units, i.e. the processor can be oversubscribed in a similar way that a CPU core with two instruction streams can. Rather than having space for two threads, each GPU multiprocessor – i.e. the hardware component responsible for executing thread groups – can have space for anywhere between 1024 and 2048 threads. Because the threads can all simultaneously store their state in the multiprocessor, i.e. they can all simultaneously be *resident* on that multiprocessor. This allows the GPU to effectively hide the latency – primarily that induced by memory accesses – by context switching at zero cost between threads; given sufficiently many threads, it is statistically likely that on any given cycle, one of the thread groups will be able to perform an operation. The number of threads that can be resident on a given multiprocessor is bound by the register usage and the memory usage of those threads, and the so-called *occupancy* that is achieved as a result of the aforementioned factors is another important performance consideration.

Another important consideration in GPGPU programming is making effective use of memory bandwidth. Although GPUs have relatively high memory bandwidths, the fact that a large group of threads issues load and store operations at the same time presents a performance risk. GPU memory achieves high bandwidth by operating at high bus widths, i.e. by allowing for the loading and storing of large amounts of *consecutive* memory. The latency of these individual loads and stores, however, is comparable to the memory subsystems of CPU architectures.

As a result, serialising one memory operation for each thread in a thread group risks ruining performance. Many GPU architectures resolve this issue by *coalescing* memory accesses, i.e. by combining memory accesses that fit within the bus width. In the best case scenario, all accesses issued by a thread group can be coalesced into a single access; in the worst case scenario, each memory access has to be issued separately, incurring significant overhead. The degree to which accesses can be coalesced – which is greatly impacted by the layout of data, as well as the way in which that data is accessed – is an important performance-relevant consideration in GPGPU programming.

Finally, it is worth noting that most – albeit not all – GPGPU architectures are designed such that the memory of the GPU is separate from the system’s main memory. This means that, in order to run a kernel on data, those data have to be explicitly copied from the system’s main memory to the GPU’s memory which can be a costly operation, especially if performed over a PCIe bus. The fact that GPU and CPU memory are usually disjoint raises important design decision for heterogeneous programming, i.e. programming in which the goal is to effectively utilise both CPU and GPU architectures in the same system. Indeed, decisions need to be made about *where* – i.e. on which device – a given kernel is to be performed, which is referred to as *placement* as well as in which order kernels are executed, which is referred to as *scheduling*. Both placement and scheduling are performance-critical in heterogeneous computing techniques and have been extensively studied [104]. Although novel system architectures which share memory between CPU-like and GPU-like devices are becoming more popular – the NVIDIA *Grace Hopper* architecture serves as a particularly pertinent example [105] – we foresee that placement and scheduling will remain relevant in the future.

3.3.3 Re-Convergent Evolution

The field of evolutionary biology describes a remarkable phenomenon known as *carcinisation*, described by Lancelot Alexander Borradaile as ‘the many attempts of nature to evolve a crab’ [106]: there appears to exist a strong tendency for seemingly different species to independently evolve into crab-like creatures. King crabs, for example, evolved crab-like bodies independently from the so-called true crabs, and the same has happened for several other species. Following the preceding parts of this section – and, not unimportantly, the marketing materials disseminated by GPU manufacturers – it is tempting to think that CPUs and GPUs are the computational equivalents of, say, panthers and salmon: completely

different species with fundamentally different properties. Although this was not an inaccurate assessment in the early 2000's, the reality is that – in a process not dissimilar to the convergent evolution of crab-like species – modern CPUs and GPUs have evolved to be surprisingly similar. These architecture familiar are more akin to, say, panthers and leopards. To round off our discussion on GPU architectures, we will therefore proceed to look at the commonalities between CPU and GPU architectures, and to dispel some of the myths surrounding them.

We posit that a lot of the misconceptions around the differences between CPU and GPU architectures stem from the choice of language used when describing these architectures, principally by GPU manufacturers themselves. In particular, we posit that the word ‘core’ is used misleadingly between CPU and GPU architectures. NVIDIA, for example, advertises their A100 GPU as having 6912 so-called ‘CUDA cores’ [64], which makes it tempting to think that it is about 100 times more powerful than an AMD EPYC 7763 CPU with 64 cores, but nothing is further from the truth. Indeed, the NVIDIA A100 operates at a maximum frequency of 1.41 GHz and can perform two operations per core per cycle⁶ for a combined theoretical throughput of $1.41 \text{ cycle/s} \cdot 6912 \text{ core} \cdot 2 \text{ FLOP}/(\text{core cycle}) = 19.5 \text{ TFLOP/s}$. The AMD EPYC 7763, on the other hand, operates at a base frequency of 2.45 GHz⁷ and performs 32 operations per cycle for a theoretical throughput of $2.45 \text{ cycle/s} \cdot 64 \text{ core} \cdot 32 \text{ FLOP}/(\text{core cycle}) = 5.0 \text{ TFLOP/s}$. Although the performance difference – a factor four – is significant, it pales in comparison to the factor 100 that we would expect from the core counts alone.

The core idea here is that the cores in CPU architectures are very different from the ‘cores’ in GPU architectures. This is particularly evident from the number of instructions that these cores can perform per cycle. The AMD EPYC 7763's Zen 3 cores each have the ability to perform 32 single-precision floating-point operations per cycle⁸ whereas each CUDA ‘core’ in the NVIDIA A100 can only perform 2 operations per cycle. Thus, a single CPU core is significantly more powerful than a single GPU ‘core’. The primary reason why CPU cores are so powerful is that they incorporate data-level parallelism, and this is what motivates us to say that the designs of CPUs and GPUs have re-converged somewhat; GPUs are and have always been data-parallel computing devices, but CPUs are increasingly moving

⁶The NVIDIA A100 can perform two floating point operations per cycle because it can perform a fused multiply-add operation, i.e. it can compute $d = a \cdot b + c$ in a single operation rather than in two.

⁷This comparison favours the GPU, because we compare the base frequency of the CPU against the maximum frequency of the GPU.

⁸This number is given by multiplying the number of FMA execution units (2), the vector widths of these units (8), and the number of operations performed per FMA instruction (2).

in the same direction with larger vectors and – equivalently – more SIMD lanes. In terms of computational power, GPU ‘cores’ are closer to SIMD lanes in CPUs than they are to entire CPU cores.

Arguably, many of the performance advantages that GPUs provide come from their *programmability*. The technical definition of programmability simply denotes the ability for a system to be programmed such that it reliably reaches a given state, but this is not particularly relevant in this day and age; indeed, even the cheapest microcontrollers available on the market today can be programmed to perform virtually any kind of operation. We posit that a more useful definition of programmability is the ability of a system to be programmed such that it reaches a given state *efficiently*. Under this definition, CPU cores are highly programmable: advanced compilers are able to correct for many performance anti-patterns that programmers might introduce, and the intricate instruction-level parallelism inside of the cores further improves the performance of what might be otherwise suboptimal code. On the other hand, SIMD lanes in modern CPUs are barely programmable at all: compilers often struggle to generate efficient SIMD code, and there is little to no in-core infrastructure that can rescue code that doesn’t use SIMD lanes efficiently. A GPU ‘core’, then, is a device with the computational power of a traditional SIMD lane, but which is far more programmable. For example, the zero-cost context switching that GPUs provide allows threads to hide latency incurred by other threads very efficiently. The ability of GPUs to conditionally execute instructions also makes them much more programmable than SIMD lanes, although this is not without performance penalties⁹. Finally, the ability of GPU threads to perform scattered loads allows them to be easy programmed in a way similar to CPU threads.

We are now equipped with sufficient understanding of massively parallel architectures to understand the challenges involved in programming them. We close this section by mentioning a fundamental paper on the topic of GPU performance versus CPU performance, which was written by by Lee et al. [107]. They posit that the performance difference between CPUs and GPUs is more likely to be in the single-digit factors than the commonly cited factor 100, which we have discussed earlier in this chapter. In this thesis, we aim to develop track reconstruction algorithms that outperform equivalent programs which run on CPUs by a similar factor, but this will not be easy: despite the fact that modern GPUs are highly programmable, performance pitfalls remain myriad and a nuanced approach must

⁹Interestingly, AVX-512 introduces conditional execution for individual SIMD lanes, further narrowing the gap between CPUs and GPUs.

be taken to avoid them.

3.4 Programming Models

To conclude this brief discussion of (massively) parallel computing, we will briefly discuss the ways in which GPUs can be programmed. After all, even the most powerful processor is useless unless it can be programmed to perform useful computations. We will only discuss programming models which can be used to program GPUs directly, i.e. we will not discuss technologies like MPI which are used in distributed computing, even if those distributed systems may contain GPUs. We will discuss programming models which are designed for GPUs in particular, as well as programming models which support a broader range of hardware.

GPU-specific programming models include the CUDA programming model, which we have discussed previously in this chapter. The CUDA programming model is widely adopted and is based on the C and C++ programming languages. CUDA works by separating the compilation of a translation unit into a host part – to be executed on a CPU – which is compiled using a commodity compiler like GCC or CLANG and a device part – executed on the GPU – which is compiled using a proprietary compiler which produces machine code that can be executed on NVIDIA GPUs. In CUDA, programmers are required to write code in a way that strongly resembles sequential code, i.e. the program is written from the perspective of a single core. The resulting *kernel* is then executed in parallel implicitly. The parallelism model is such that individual threads are grouped into *blocks* – which map onto multiprocessors – which are, in turn, part of a *grid* – which map onto entire GPUs [97]. A more recent competitor to CUDA is HIP, which allows CUDA-like programming for AMD GPUs [108]. Further ways to program GPUs include OpenGL and its successor VULKAN. Although both OpenGL and VULKAN are designed primarily for graphics workloads [109], they allow the user to design custom compute shaders which can be used to achieve similar results to CUDA and HIP.

A more recent development has been that of programming models which can target a broad range of hardware. Often, this includes GPUs and multi-core CPUs. Some programming models also support the programming of FPGA devices, although this is a much less proven strategy. Prominent mixed-target programming models include OPENMP and OPENACC, which extend the C, C++, and Fortran programming languages with pragmas which can be used to direct the compiler to

make code transformations towards parallelism. In comparison to CUDA, OPENMP and OPENACC programs are written with explicit loops [110, 111]. Portability layers or libraries form another class of parallel programming models, which are usually implemented directly into an existing language. Models such as SYCL [112], KOKKOS [113], ALPAKA [114], and the parallel component of the C++ standard library are all implemented in C++; they provide abstractions for building parallel programs, but they do not fundamentally extend the programming language. Finally, there are dedicated programming languages for mixed-target parallelism, including FUTHARK [89] which is an array programming language, CHAPEL [115] which features many parallel primitives, and OPENCL [116] which offers CUDA-like implicit parallelism. Dedicated programming languages can often provide useful abstractions which make programming easier, while also proving competitive performance [117].

Beyond general-purpose computation, there are many fixed-function parallel libraries. The motivation behind this is that there exists a set of kernels which are both widespread in various computing applications and also highly demanding in terms of performance. Amdahl's and Gustafson's law dictate that providing high-performance implementations of these specific algorithms can bring great speed-up in many applications. Fixed-function parallel libraries exist for a broad range of algorithms, including fast Fourier transforms, linear algebra, and neural network inference. Finally, it is possible that, in the future, sufficiently complex compilers will be able to automatically parallelise any given program – even a program not explicitly parallelised by a human – for a given target, but we are not aware of the existence of any such compilers at this time.

4

State-of-the-Art Algorithms for Track Reconstruction

*You can't connect the dots looking forward;
you can only connect them looking
backwards. So you have to trust that the dots
will somehow connect in your future.*

— Steve Jobs
(Co-founder of Apple Inc.)

In the previous chapters, we have examined the track reconstruction problem, as well as the massively parallel hardware on which we aim to solve it. Now, we move towards *solutions* to that problem. In this chapter, we describe the state of the art in track reconstruction algorithms in order to answer Research Question 1. Through this examination, we identify the performance-related challenges which these algorithms present in a massively parallel world so that we can solve those problems in later chapters. We categorise the algorithms involved in track reconstruction according to the *13 dwarves of parallel computation* [21] in order to illustrate their general structure.

Some parts of this chapter are based on the following publication:

- The ATLAS Collaboration and Stephen Nicholas Swatman. ‘Software Performance of the ATLAS Track Reconstruction for LHC Run 3’. In: *Computing and Software for Big Science* 8.1 (Apr. 2024). ISSN: 2510-2044. DOI: 10.1007/s41781-023-00111-y

4.1 Introduction

The days of analogue particle detection using nuclear emulsions and cloud chambers – which we describe briefly in Chapter 1 – are now well behind us. A switch

to digital data collection has allowed collision rates previously unheard of; indeed, the Large Hadron Collider is capable of colliding proton bunches at an incredible rate of 40 MHz. Unfortunately, digitalisation in particle physics has brought the same trade-off that it brought to, say, photo- and videography: it allows for the acquisition and storage of large amounts of data, but it requires this data to be *discrete*. Digital imagery captures light at a fixed number discrete points known as pixels. Traditional film stock, on the other hand, captures light across a continuous array of light-sensitive molecules¹; it is the *continuous* nature of film that allows it to capture exquisite amounts of detail, as evidenced by the 2012 remastering of *Star Trek: The Next Generation* – shot in 1987 on 35 mm film – which rendered the voyages of the starship *Enterprise* in resolutions that are impressive even by today's standards.

The loss of continuity in data is rarely a problem in imaging: the human brain has the remarkable ability to view a grid of coloured pixels and to interpret it in real-time: even a very-low-resolution video with 320×240 pixels can convey a story – even though, these days, it may be met with some complaints about the lack of sharpness. In particle physics, the loss of continuity poses a far greater challenge, however. When the trajectory of a particle is captured in a detector, its curvature in a magnetic field – see Chapter 2 – can be used to derive information about its charge and energy, both of which are crucial to understanding the behaviour of the particle and the collision event as a whole. Thus, discrete data collection requires the so-called *reconstruction* of the continuous trajectories, which is referred to as *track reconstruction*.

In this section, we discuss the state-of-the-art in track reconstruction, which primarily encompasses algorithms for CPU-like architectures, although some research on massively parallel architectures is available. The track reconstruction process – and, as a result, the work which we do in this thesis – is preceded by the process of gathering data. When processing empirical data, this involves the read-out electronics, zero suppression, and so forth. In simulated data, this involves the simulation of particles and the way those particles interact with the detector. We consider these preceding steps to be outside the scope of this thesis, however; the gathering of data is far more akin to electrical engineering than it is to high-performance computing and simulation is a fiendishly complex topic about which plenty theses could be and have been written. We consider our input

¹It is worth noting that film, examined at a sufficiently fine level of detail, is of course also discrete: such is the nature of the molecules of which it is composed. Furthermore, film does not necessarily possess infinite resolution: a roll of FUJICHROME Velvia 50 Professional can resolve approximately 160 lines per millimetre of film before they blur together due to physical and chemical effects [118].

data to be data about which sensors in the detector were activated, and to which degree. Our target output data are the particle tracks. From there, further analysis can be performed to find e.g. the so-called *vertices* at which particles originate, but we consider this more physics-oriented data processing beyond the scope of our work as well. With these boundaries in place, the track reconstruction pipeline can be roughly broken down into three steps: preprocessing, track finding, and track refinement, all of which we will discuss here.

4.2 Existing Implementations and Challenges

Software-based track reconstruction has been an important topic in high-energy physics experiments for decades and, naturally, there have been dozens of implementations for different detector geometries and different computing architectures. In this section, we will explore some contemporary track reconstruction software packages, the metrics that determine how effective these packages are, and the challenges that they will face in the future.

4.2.1 Libraries and Frameworks

The Large Hadron Collider has two general-purpose experiments, each with their own software packages that facilitate track reconstruction; the ATLAS experiment has the so-called ATHENA package [19] and the CMS experiment uses software known as CMSSW [119]. The smaller experiments on the LHC also use dedicated software packages, and the same goes for other particle physics experiments around the world. Both ATHENA and CMSSW consist of millions of lines of code and contain many packages which perform different functions, only some of which relate to track reconstruction. Furthermore, many of these software packages have grown organically over long periods of time; the ATLAS software – which, at that time, was not yet known as ATHENA – dates back to as early as 1996 [120] and has evolved from consisting of mostly FORTRAN code to a C++-based software package. It is virtually inevitable that software developed over such a long period of time will decay to some degree or another, leading to inefficiencies or difficulties in maintaining the software [121]. Furthermore, many existing software packages were designed for experimental environments which are much less demanding than the ones encountered today, let alone in the future.

The A Common Tracking Software (ACTS) project is an attempt to resolve many of the existing problems with tracking performance by providing an experiment-

agnostic software package developed according to modern programming practices [122]. The ACTS project is also designed to alleviate pressure on smaller experiments; although experiments such as ATLAS have sufficient person power to develop massive code bases for track reconstruction, this luxury is not afforded to experiments with dozens rather than thousands of members. Such experiments can use the ACTS project to solve many or all of their track reconstruction problems with relatively little effort, as has been shown in e.g. the sPHENIX [123], FASER [124], and ePIC [125] experiments. Finally, the ACTS project is designed as a research and development platform, allowing for the rapid development and evaluation of novel track reconstruction algorithms and implementations thereof. In this thesis, we will make extensive use of the ACTS project to support our own research.

It is worth noting that there have been significant efforts towards massively parallel track reconstruction in the past. Much of this efforts has gone towards the development of massively parallel implementations of specific sub-problems, principally the Kálmán filter (see Section 4.4.3) as in the works of Ai, Mania, Gray, Kuhn and Styles [126] and Cerati, Giuseppe et al. [127]. Other efforts have been towards the development of massively parallel track reconstruction software for specific experiments; examples include developments by Gorbunov et al. [128, 129] in the ALICE experiment and by Cámpora Pérez, Neufeld and Riscos Nuñez [130] in the LHCb experiment. The LHCb experiment has been especially prescient in the development of their ALLEN framework [131] which implements important features such as co-processor scheduling. The developments of PATATRACK by Bocci, Innocente, Kortelainen, Pantaleo and Rovere [132] in the CMS experiment has also been a significant step towards the exploitation of massively parallel processors in track reconstruction. Other work, such as that by Amerio et al. [133] has provided massively parallel algorithms constrained to trigger environments. The aforementioned work has significantly advanced the field of computing in high-energy physics, but we argue that none of these solutions are so general that they can function in any experiment and in any environment, leaving important work to be done, especially in the ACTS project. Furthermore, we are not aware of any conclusive proof that the aforementioned approaches maximally exploit the hardware on which they run, which indicates that there may be room for further improvements.

4.2.2 Efficacy Metrics

Track reconstruction algorithms are usually evaluated according to three metrics of efficacy. The first is the number of tracks that are correctly reconstructed, i.e. the number of particles for which at least one track was found. In the field of particle physics, this is often referred to as the *efficiency* of the algorithm, but we consider this to be somewhat confusing terminology. Instead, we shall refer to this metric using the more precisely defined term *recall*. A higher recall is considered to be better. The second metric is the *fake rate*, which describes the number of tracks that are found which do not correspond to a true track. Fakes are type II errors and, as such, the fake rate corresponds to the type II error rate. A lower type II error rate is considered to be preferable. The third and final metric is the so-called *duplicate rate*, which is the total number of true positive tracks divided by the total number of particles for which at least one track was found. The duplicate rate indicates the number of redundant tracks that were found which has extra-functional side-effects in terms of computational efficiency and disk storage, so a lower duplicate rate is considered better. Note that the aforementioned metrics taken together are referred to as the algorithm's *performance* in the field of physics, but we find that this is easily confused with computational performance; therefore, we will instead use the term efficacy.

4.2.3 Future Challenges

At the time of writing, experiments around the world are successfully reconstructing collisions with homogeneous CPU-based architectures. However, there are plans – some theoretical, some under way – to upgrade colliders and the experiments thereupon to increase the number of collisions significantly. This happens, for example, due to an increased ability to squeeze protons into a smaller space. In ATLAS, it is predicted that the average pile-up $\langle\mu\rangle$ will increase from around 50 during the run 3 operating period (2022–2025) to up to 140 in the run 4 operating period (2029–2032) and to 200 in the run 5 operating period (2035–) [18]. Similar increases are expected in other experiments such as CMS. We recall that the computational complexity of track reconstruction increases superlinearly with $\langle\mu\rangle$ and, as such, the aforementioned upgrades present a significant computational challenge. Even more challenging are designs for future colliders such as the Future Circular Collider (FCC) [134]. Concurrently, detectors are also becoming more precise and are therefore producing more data. The ATLAS Inner Detector (ID), for example, is being upgraded to the Inner Tracker (ITk) which will increase the number of

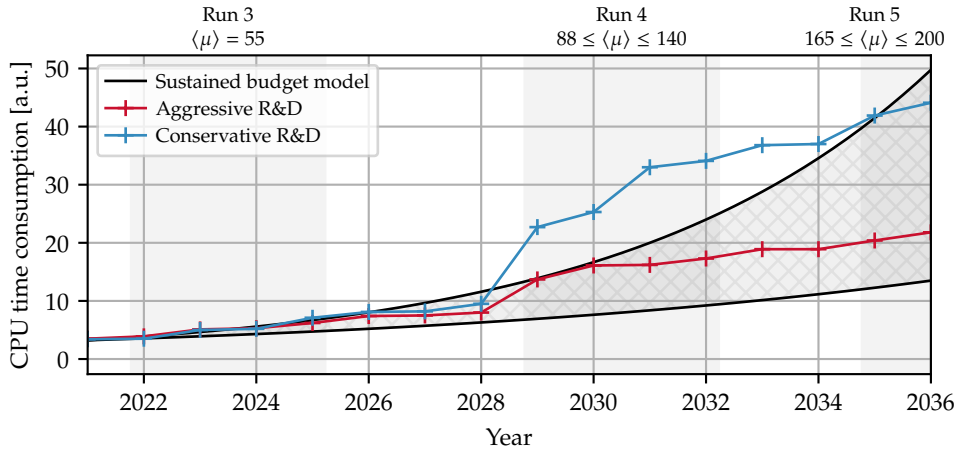


Figure 4.1: Projected CPU time requirements of the ATLAS experiment under aggressive and conservative research and development programmes into software performance compared to a sustained increase in hardware availability and performance between 10 % and 20 % per year due to budget increases and hardware improvements. Plot adapted from the ATLAS Software and Computing HL-LHC Roadmap [18].

data channels from approximately 100 million to 5 billion [135]; naturally, this will massively increase the volume of incoming data and the amount of computation required to process that data.

Figure 4.1 shows an estimate of the amount of processing power required to meet the data challenges posed by the Large Hadron Collider over the coming twelve years. It is clear that sitting idly at the sidelines hoping that increases in hardware performance will permit current approaches to keep working is a plan doomed to fail; indeed, an aggressive programme of research and development is necessary in order to enable physicists to process the data of the HL-LHC and future experiments. Aggressive research and development, in this case, encompasses a broad range of efforts by many people of different skill sets, but is also relies – quite strongly, indeed – on the adoption of novel hardware architectures including massively parallel ones. In this thesis, we aim to show that such architectures can, in fact, be adopted in high-energy physics, but it is clear that this will not be a ‘free lunch’. In the remainder of this chapter, we will explore the challenges involved in developing an experiment-agnostic track reconstruction toolkit that is able to efficiently leverage massively parallel processors such as GPUs.

4.3 Preprocessing

As data leaves the detector – or, indeed, a simulation – it carries remarkably little information that is relevant to physics. In fact, the data that is received is little more than the amounts of charge that were measured in different parts of the detector. The preprocessing stage of the track reconstruction, therefore, is characterised by a surprising lack of domain-specific problems: most of the operations contained therein are common across many fields of computational science, although the specific parameters of these problems add additional challenges.

4.3.1 Connected Component Analysis

As described in Section 2.4, a charged particle passing through a silicon detector will deposit a small but measurable charge in the detector material. The interpretation of this charge lies at the core of the tracking chambers of many experiments. Unfortunately, the deposition of charge does not always occur in a point-like fashion, i.e. the passing particle does not activate a single sensor but – potentially – a neighbourhood of sensors. This effect – known as charge sharing – is caused by the non-zero thickness of the detector; because the detector consists of a three-dimensional charge sensitive volume and a charge sink, it is possible for a particle passing through the detector at a non-right angle to deposit charge into multiple sensors as it traverses the depth of the detector. An schematic example of how particles at right and non-right angles deposit charge onto silicon detectors is shown in Figure 4.2. The sharing of charge between neighbouring sensors is further exacerbated by a variety of physical effects [136], but the details of these effects are outside the scope of this thesis.

Charge sharing prohibits us from assuming a one-to-one relationship between charge depositions – which we will refer to as *hits* – and particle–detector interactions which we will refer to as *measurements*. Doing so would both vastly overestimate the number of such interactions (thereby increasing the combinatorics inherent in the track reconstruction problem, as we will see in Section 4.4.3) and reduce the accuracy of the estimates of where those interactions took place. It is important, therefore, to aggregate multiple hits produced by a single particle into a single measurement. This process canonically takes place in two steps: first, hits are clustered, after which they are aggregated into their weighted centroid according to Equation 4.1, where x_i and y_i are the two-dimensional local coordinates of hit i , and w_i is the weight of that hit, determined by the amount of charge deposited. Note that Equation 4.1 assumes that all hits belong to the same cluster, i.e.

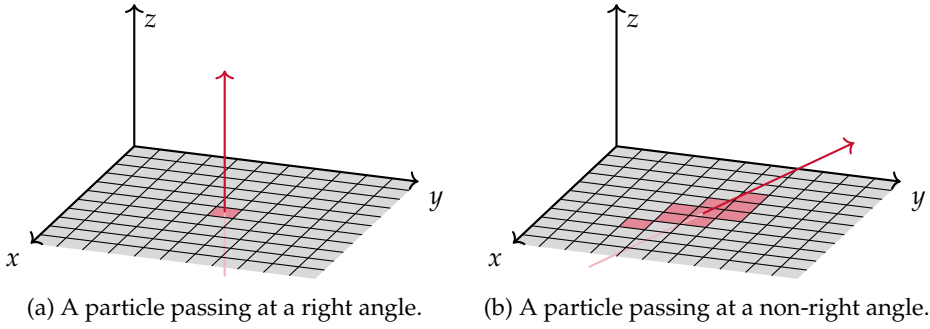


Figure 4.2: Schematic overview of two particles passing through a detector surface – aligned with the horizontal plane – at right and non-right angles.

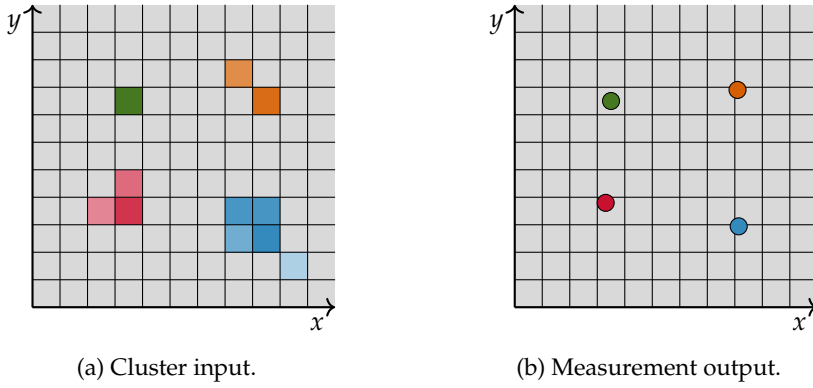


Figure 4.3: An illustration of the measurement creation process, which computes the weighted centroids of clusters. Each colour matches a single cluster, and darker colours imply higher weights.

this aggregation is performed for each cluster separately. For each measurement, we also record the degree of uncertainty about the exact crossing location of the particle which will be necessary for later steps in the algorithm chain.

$$f((x_1, y_1, w_1), \dots, (x_n, y_n, w_n)) = \left(\frac{\sum_i^n x_i w_i}{\sum_i^n w_i}, \frac{\sum_i^n y_i w_i}{\sum_i^n w_i} \right) \quad (4.1)$$

The problem described in this section is a so-called 8-connectivity Connected Component Analysis (CCA) problem. More specifically, the problem consists of a so-called Connected Component Labelling (CCL) problem followed by an aggregation of the connected components – or, as we refer to them in the context

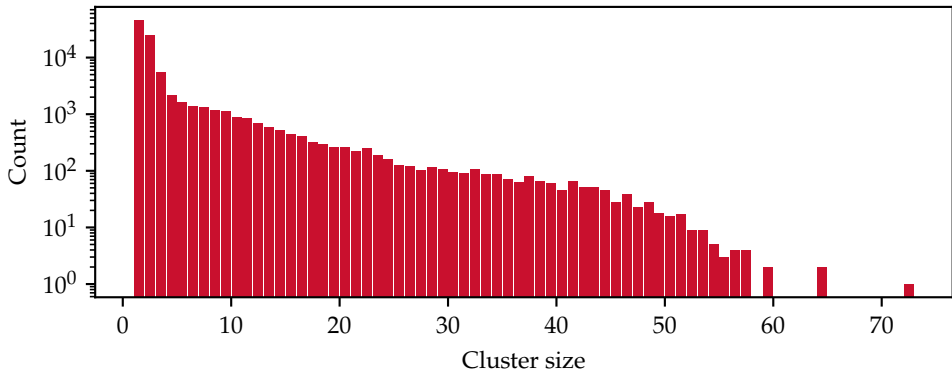


Figure 4.4: Histogram of the sizes of clusters in a simulated event in the TRACKML detector [138] with $\mu = 200$.

of track reconstruction problems, clusters – which were labelled in the preceding step. Connected component labelling and analysis are two of the most broadly studied problems in image and graph processing. There is, therefore, a wealth of literature available on how to solve such problems on a variety of computation devices [137].

Unfortunately, the connected component analysis that is to be performed in the context of track reconstruction has one property that is uncommon in other fields, which is that it operates on *sparse* data. The target density – i.e. the fraction of pixels that carries non-zero charge – in the ATLAS Inner Detector is approximately 1% [139]. Furthermore, the data is provided sparsely, canonically in the ordered Coordinate List (COO) format [140]. This sparseness discourages the naive use of dense CCL algorithms that are common in image processing, as this would require the reification of the sparse data into a large and mostly empty dense array. It is also worth noting that most clusters are very small; as shown in Figure 4.4, the vast majority of clusters contain only a handful of pixels.

In order to efficiently perform CCL operations on sparse data, the SPARSECCL [141] algorithm was developed. In fact, SPARSECCL was developed specifically for applications in high-energy physics and is commonly used in a variety of experiments [142]. It has been shown that SPARSECCL can outperform dense algorithms significantly for data with levels of sparseness similar to the data in high-energy physics experiments. SPARSECCL is a strongly sequential algorithm as evidenced by the fact that the original authors were unable to improve its performance using data parallelism [141], but this problem can be easily negated on MIMD architec-

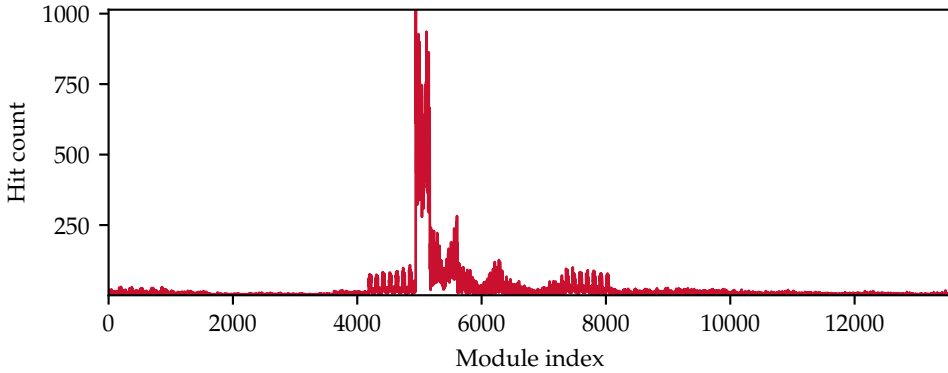


Figure 4.5: The number of hits on the different modules in a simulated event in the TRACKML detector with $\mu = 200$.

tures by relying on the crucial fact that clusters can never cross module boundaries. In other words, each of the thousands of modules in a detector can be processed in a batched fashion, which allows embarrassingly parallel execution on multi-core CPUs.

The connected component analysis step poses one of the first – and indeed one of the greatest – challenges in the implementation of track reconstruction software for massively parallel hardware: *workload imbalance*. Two modules in a detector can have vastly different numbers of hits; Figure 4.5 shows an example of how the number of hits per module can differ in an event with $\mu = 200$ for the TRACKML detector [138]. Due to the lock-step nature of massively parallel architectures, naive implementations would suffer from the fact that all threads in a thread group would have to wait for the thread with the largest amount of work – i.e. the largest number of hits – to finish, which may severely impede performance. We will explore ways to model and mitigate thread imbalance like this in Chapter 7. In terms of dwarves, we classify this step of the track reconstruction pipeline as ‘Structured Grids’ due to the lattice-like structure of modules, ‘Sparse Linear Algebra’ due to the similarity of the data to sparse matrix data, and to a lesser extent ‘Graph Traversal’, as we can treat the sparse data as a graph; we will explore this further in Chapter 8. The creation of measurements from clusters is a simple ‘MapReduce’ problem.

It is possible, albeit unlikely², that two or more particles cross a sensitive surface in such close proximity that their energy deposits form a single cluster by the

²Barring, for example, particles that decay very close to the sensitive surface

8-connectivity definition. There are at least four distinct strategies for dealing with such scenarios. The first strategy is to produce a single measurement from such a cluster and to assume that it was produced by a single particle, which would potentially reduce the efficacy of the track reconstruction algorithm. The second strategy is to produce a single measurement but to assume that any single measurement may be linked to more than one particle, which risks increasing the combinatorics of track finding (Section 4.4). The third strategy is to produce multiple measurements by splitting the cluster during the CCL process, which also risks incurring more demanding combinatorics. Finally, the fourth strategy is to produce a single measurement but to retain the cell structure of it so that it can be split later if there is evidence that the measurement belongs to multiple particles. The latter strategy is used in the ATLAS experiment, where small neural networks are used to split clusters; this has been shown to increase the efficacy of track finding, but also incurs significant computational overhead [143]. In this thesis, we do not consider multi-particle clusters, although efficient algorithms for splitting them in massively parallel environments could be an interesting avenue of future work.

4.3.2 Spacepoint Formation

Following the clustering step described in the previous section, we are left with the weighted centroids of the activations on the detector surfaces. These surfaces, however, are two-dimensional and distributed around the detector at different positions and orientations. We refer to these points as *bound* to a given surface. In order for these points to have meaningful three-dimensional coordinates, they must be converted into the *global* frame; these global coordinates are referred to as *spacepoints*. The aforementioned conversion – known as *spacepoint formation* – is achieved through a simple $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ coordinate transformation, which can easily be achieved through the multiplication of 4×3 homogeneous matrices – 3×3 in a compressed format – by the coordinate vectors. This process by itself is embarrassingly parallel on both MIMD and SMT architectures.

The primary challenge in spacepoint formation involves the retrieval from memory of the transformation matrices associated with each measurement. Thankfully, the transformation matrices are quite small – 36 B for single-precision formats – and can be accessed in constant $O(1)$ time if the matrices are stored in such a way that the detector surface identifiers serve as array indices. The storage of such matrices is usually handled by detector description libraries such as the ones

described in Section 2.4. We classify the spacepoint formation step as following the ‘MapReduce’ pattern for its embarrassingly parallel transformation mapping, as well as the ‘Dense Linear Algebra’ pattern as it involves a large number of matrix multiplications.

4.4 Track Finding

Following the preprocessing stage, we have three-dimensional points in a detector volume which represent the locations at which particles were detected. We do not know – however – to which particles these points belong; unfortunately, particles do not leave a metaphorical business card in the detector which can be used to identify them. It is up to us, therefore, to determine which points belong to the same track, and this process is known as *track finding*; it is here that our computation will first start to be governed by the behaviour of particles as determined by the laws of nature.

4.4.1 Seed Finding

A naive strategy to find tracks would be to simply explore all possible combinations of points. This strategy, however, becomes infeasible for any non-trivial input; tracks in real-world experiments usually consist of anywhere between three and twenty spacepoints, and the number of spacepoints in a single event can easily reach into the tens of thousands. For this reason, it is considered canonical to produce so-called *seeds*. Seeds are combinations of the minimum number of spacepoints which can be used to define a track, after which the seed can be expanded to include additional spacepoints if necessary. In this way, the combinatorial explosion in the processing of the data can be minimised.

The vast majority of particle physics experiments is designed such that the minimum number of spacepoints that describes a track is three, which has to do with the magnet system in these experiments. As described in Section 2.4, virtually all such experiments are permeated with a powerful magnetic field, as the curvature of a charged particle track in a magnetic field of known strength allows us to estimate its kinetic energy. This magnetic field is usually approximately homogeneous and runs parallel to some vector which we will – as is convention in the ATLAS experiment – call z ; we will assume that x and y form an orthogonal basis alongside z . This construction gives us three important corollaries. Firstly, since the Lorentz force (see Equation 2.6) always imparts a force perpendicular to

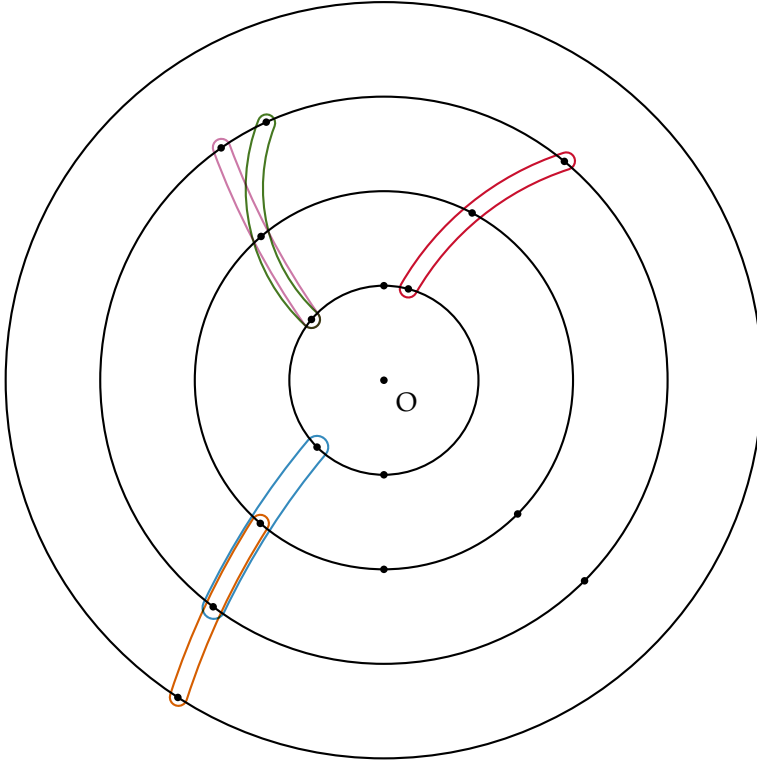


Figure 4.6: An example of seed finding with 16 spacepoints (denoted by circles) around the collision point O in the x - y plane. In total, five seeds are found as denoted by the contours around the spacepoints. Note that spacepoints can be shared by multiple seeds.

the magnetic field, it never accelerates a particle along the direction of that magnetic field; therefore, the velocity of the particle in the z direction is constant. Secondly, following the aforementioned, any particle with sufficiently high momentum such that the distance $r = \sqrt{x^2 + y^2}$ from the origin of the x - y plane – where we assume the collision to happen – increases approximately linearly in time will exhibit a linear correlation between its distance along the z axis and its radius r . Thirdly and most importantly, the magnetic field causes the particle to move in a circle on the x - y plane as it continuously bends the trajectory of the particle.

Although research is ongoing to allow experiments to measure the time at which a particle passed through a detector [144], this is not yet commonplace. Therefore, the first of the aforementioned corollaries does not directly aid in the description of particle tracks. Since the time components in the correlation between the particle

radius r and its distance in z cancel out, however, the linear correlation between these (known) values holds. Thus, we know that the track of a particle can be partially described by a line – or, equivalently, two spacepoints – in the r - z plane. Finally, the behaviour the track in the x - y plane can be described by a circle, which in turn is equivalent to three spacepoints. Combined, a particle can thus be described by three arbitrarily chosen points, given that they are colinear in the r - z plane. Finding such seeds is the task of a so-called *seed finding* algorithm.

In real-world particle physics experiments, seeds are additionally constrained. Which constraints are relevant and which are not depends on the geometry and design of the experiments, but a selection of common constraints on seeds is given below:

1. The seed should be approximately linear in the r - z plane as discussed earlier.
2. The intersection of the line that describes the seed in the r - z plane should intersect the z -axis, i.e. should reach the presumed collision point at $r = 0$, within a certain range in z . In the ATLAS experiment, the standard deviation of the luminous region, i.e. the region where collisions happen is around 30 mm to 40 mm and decreasing every year [41].
3. The circle described by the seed in the x - y plane should have a minimum radius, which corresponds directly to the momentum of the particle. Low-momentum particles are usually considered to be less interesting, are far more numerous, and are significantly harder to reconstruct.
4. The aforementioned circle should approach the z -axis, i.e. the assumed collision point, within a certain distance.
5. Two consecutive spacepoints in the seed should differ in their angle ϕ around the z -axis by a limited amount. This constraint correlates strongly with constraint Item 3, but additionally ensures that two points on opposite sides of the z -axis cannot form a seed.
6. Two consecutive spacepoints should appear on two adjacent sensitive surfaces, i.e. there should be no surface between them on which a spacepoint has *not* appeared. As this query is computationally expensive and because most experiments place surfaces at fixed positions along the r -axis, this constraint can be cheaply approximated by imposing a limit on the r -coordinate of two consecutive spacepoints.
7. The line in the r - z plane described by two spacepoints should not run too close to the beampipe, where the performance of the detector is much lower.

The so-called *pseudorapidity*, defined as $\eta = \text{arctanh}(\vec{p}_z/|\vec{p}|)$ (where \vec{p} is the momentum of the particle) captures this [23]; an upper bound on the value of $|\eta|$ ensures that tracks are only found in relevant regions.

8. Spacepoints should be limited by a maximum r value, as well as a maximum value of $|z|$ because, in most experiments, it is worth finding seeds only in the innermost detectors which provide the highest precision and momentum resolution, i.e. the ability to accurately determine the momentum of a particle.

It is worth noting that the aforementioned constraints reduce the ability to reproduce so-called secondary particles, i.e. particles which are not created directly in the luminous region. This includes particles that are created in decays after the primary collision events. The reconstruction of such particles is of lower concern than reconstructing primary particles, however, and is – as such – not covered in this thesis. Similarly, low-energy particles are often excluded to the significant difficulties involved in the reconstruction of their tracks. In particular, particles with such low energies that the circular projection of their trajectory lies entirely inside of the detector, referred to as *spinners*, present a great challenge in track reconstruction and they are, as such, usually omitted from the process. It is worth noting that the parameters to the aforementioned constraints can be autotuned to increase performance or efficacy [145]. Figure 4.6 shows a simple example of seed finding in the x - y plane which demonstrates some of the aforementioned constraints.

State-of-the-art seed finding algorithms focus on increasing performance by decreasing combinatorics in two primary ways. Firstly, such algorithms often break the problem of finding triplets of spacepoints down into a first step of finding pairs of points, followed by the merging of pairs which share a common spacepoint. Although this approach technically increases the complexity of the process from $O(n^3)$ to $O(n^4)$, where n is the total number of spacepoints, the constants involved allow this approach to outperform a naive triplet finding strategy. Indeed, constraints 2 and 5 to 8 can be applied to pairs of spacepoints rather than triplets and can, thereby, help to reduce the number of candidate pairs.

The second approach taken to increase the performance of seed finding is to bin spacepoints according to their location in ϕ - r - z space. This allows the algorithm to reject candidate spacepoints not by fetching them from memory and then checking the constraints but rather by not fetching those spacepoints at all. Constraints 5, 6 and 8 can be implemented in a weaker form by selecting spacepoints only in specific bins. That is to say, the presence of a spacepoint in a candidate bin does not guarantee that the spacepoint meets the corresponding criteria, but any spacepoint

found in a non-candidate bin is guaranteed *not* to meet those criteria. The geometry of the detector under study as well as the parameters of the particles for which tracks are to be reconstructed govern the degree to which a binning-based strategy can reduce combinatorics and – as a direct result – increase performance.

The seed finding problem is challenging to categorise according to the thirteen dwarves of parallel computing, as it exhibits similarities to several of them, but corresponds very strongly to none of them. We posit that seed finding resembles ‘N-Body’ problems, because it features interactions between many discrete points, and because these interactions can be culled on e.g. distance between them. Furthermore, seed finding exhibits similarities to the ‘Dynamic Programming’ dwarf as it is used to construct larger tracks from smaller triplets or even doublets of spacepoints. Finally, we propose that there are similarities to ‘Branch-and-Bound’ kernels due to the fact that spacepoints can be compatible with multiple other spacepoints, and that doublets and triplets can be pruned during the process. Challenges in implementing massively parallel seed finding emerge primarily to due irregular and imbalanced workloads; the number of spacepoints per bin can vary significantly, as can the number of candidate spacepoints that can form doublets. Similarly, the number of compatible doublets for a given origin doublet can also vary significantly. In massively parallel environments, care will have to be taken to ensure that this work is balanced to the greatest possible extent.

4.4.2 Track Parameter Estimation

Following the creation of our seeds, our goal is to extend those seeds to contain whichever additional spacepoints would fit the existing seeds. This extension is performed using a numerical propagation method, which is to say that the movement of the particle is iteratively simulated and any additional spacepoints which are sufficiently close are added to the track. In order to perform this propagation, however, we must have an estimate for the behaviour of the particle; to find such estimates is the task of the *track parameter estimation* algorithm, which finds suitable parameters for every seed. Principally, each seed can be described according to the sensitive surface on which the first spacepoint lies, and a set of five *bound* parameters, so named because they are bound to a given surface. These parameters are the local positions on the surface l_0 and l_1 , two angles ϕ and θ which describe the angle at which the particle passes through the surface, and the $q/|\vec{p}|$ parameter which captures the charge (positive or negative) of the particle, as well as its absolute

momentum and describes the curvature of the track³. As mentioned before, some experiments employ detectors with timing capabilities; these experiments have a sixth parameter t which describes the time at which the seed starts. It is worth noting that tracks can also be described in a free state, i.e. not bound to a surface, using three positional parameters l_0 , l_1 , and l_2 as well as three angles ϕ , θ , and ψ in addition to the $q/|\vec{p}|$ parameter and, optionally, t . A free track is thus described by seven or eight parameters, but we will not employ this representation in this thesis. Similarly, simpler incomplete track parameterisations such as presented by Karimäki [146] are also outside the scope of this work.

Although the estimation of track parameters is mathematically complex – we will not touch upon the exact calculations here – it is computationally simple, i.e. it is non-iterative, is not data-dependent, and accesses memory in predictable pattern. As such, the problem can be solved in an embarrassingly parallel fashion. We classify track parameter estimation as a ‘MapReduce’ problem, owing to the fact that it maps a single, predictable operation over a large number of input seeds.

4.4.3 Combinatorial Kálmán Filtering

In order to control combinatorics, seed finding looks for tracks with only three spacepoints. Although this is computationally efficient, the vast majority of tracks in real experiments have more than three spacepoints: anywhere between five and twenty spacepoints is commonplace. Because adding more spacepoints to a track allows us to more accurately fit parameters to it, it is imperative that we extend seeds to contain as many of such points as possible. This is canonically achieved using a so-called combinatorial Kálmán filter. We will proceed by providing a brief overview of Kálmán filters in general, which are widely used in a variety of domains, and then extend them with the combinatorial properties that are required in track reconstruction algorithms.

Generally speaking, a Kálmán filter is an algorithm that combines uncertain measurements of a process with model-based predictions about the behaviour of that process, in order to estimate of the state of that process at any given time in a way that is more accurate than either the measurements or the model-based predictions alone. As a simple example, we might imagine a vehicle that can be tracked using a satellite-based positioning system. We assume that we have an estimate of the position as well as the momentum of the vehicle – i.e. the state of

³There are many different ways of representing the curvature of a charged particle; the parametrisation described in this section is the one adhered to in the ATLAS experiment and throughout the remainder of this thesis.

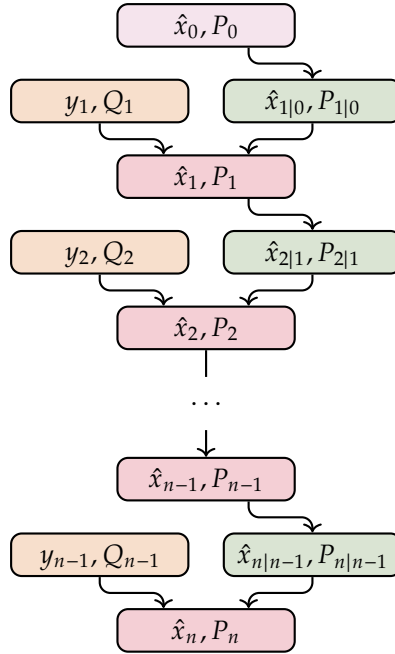


Figure 4.7: A schematic overview of the basic functioning of a Kálmán filter. The initial state is \hat{x}_0 with uncertainty P_0 . The green boxes represent predictions made from the previous state estimate using a physical model. The orange boxes represent measurements with measured state \hat{y}_n and uncertainty Q_n . The red boxes represent the filtered state estimates.

the vehicle – at a given point in time, and that we wish to gain accurate estimates of the state of the vehicle as time goes on. One solution, of course, is to periodically record the position of the vehicle using the aforementioned satellite positioning system, but we recall that these measurements are imprecise. We can improve the accuracy of those estimates by constructing a model of the behaviour of the vehicle. In this example, we can estimate the state $\hat{x} = (\vec{p}, \vec{v})$ with position \vec{p} and momentum \vec{v} of the vehicle at a future point in time from our current estimate, e.g. $\vec{p}_{n|n-1} = \vec{p}_{n-1} + (t_n - t_{n-1})\vec{v}_{n-1}$ and $\vec{v}_{n|n-1} = \vec{v}_n$. The fundamental idea behind the Kálmán filter is that the predictions given by this model and the uncertainties P of those predictions can be used to *augment* the measurement process, providing more accurate estimates of the true state of the vehicle.

A Kálmán filter works iteratively; in time step $n \geq 1$, we have access to the state estimate at the previous time step, \hat{x}_{n-1} as well as the uncertainty of that estimate, P_{n-1} . Using this estimate, we predict the state of the system at the end of

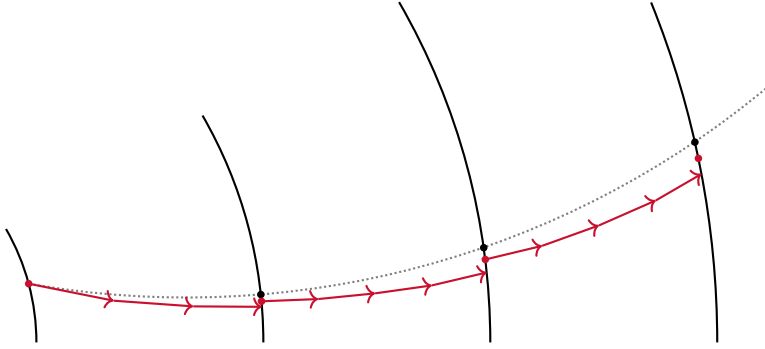


Figure 4.8: A simple example of how a Kálmán filter can be used to extend a track with additional data. The arrows indicate the physical propagation model of the particles; the black dots represent the measurements of the gray dotted track, and the red dots represent the measurements added to the track.

the current time step using the physical model of the system which gives $\hat{x}_{n|n-1}$ and $P_{n|n-1}$. Note that the physical model is to be defined by the user; modelling some physical processes can be challenging, and as we will see later on in this section, may sometimes need to be a nested iterative process. At the end of the time step, a measurement is made of the physical system which gives a state y_n with uncertainty Q_n . An update step then combines the predictions $\hat{x}_{n|n-1}$ and $P_{n|n-1}$ with the measurements y_n and Q_n to yield a new state estimate \hat{x}_n with uncertainty P_n . The new estimated state is computed according to the relative uncertainties of the prediction and the measurement; this weight factor is known as the Kálmán gain. The estimated uncertainties are given according to the covariance update equation. The aforementioned iterative process can be repeated as many times as necessary, and Figure 4.7 gives a graphical representation of the aforementioned process. It is also worth noting that it is assumed that the errors are Gaussian in nature, which is a valid approximation for many real-world problems [147].

Intuitively, the application of Kálmán filtering to track reconstruction is as follows. From the track estimates for a given seed, we have the ability to propagate the hypothetical trajectory through space according to the Lorentz law. As we propagate across longer and longer distances, the uncertainty of our prediction increases. Indeed, uncertainty increases due to a variety of physical effects including but not limited to (1) multiple scattering: changes in the direction of a particle as it passes through matter; (2) energy loss incurred as a particle passes through matter as described by the Bethe-Bloch formula; and (3) Bremsstrahlung:

loss of energy and therefore momentum as electromagnetic radiation is emitted by a particle [148]. Whenever another spacepoint is found inside the region defined by that uncertainty, it is a valid extension to the track. The Kálmán formalism then dictates that we can use that spacepoint to improve the accuracy and precision of our search for further spacepoints. This improvement in accuracy is beneficial not only functionally because it produces higher quality tracks, but it also reduces the number of spacepoints we need to consider, thereby improving the combinatorics of track reconstruction. Figure 4.8 is an illustration of how Kálmán filtering can be used to find spacepoints that are compatible extensions to an exiting track.

So far, we have discussed how Kálmán filtering can be used to extend seeds into tracks, but we have neglected an important part of the Kálmán formalism: the model of the behaviour of the system. We recall that this model is to be provided by the user, and should be able to produce both a prediction of the state, i.e. the position and momentum of a particle, after a given amount of time, as well as the error in that prediction. Since an initial state of the model is always known and we know that charged particles move according to the Lorentz force, the propagation of particles through space poses an Initial Value Problem (IVP): the initial state is given, and the derivative of the state is derived from the Lorentz force. Importantly, this IVP cannot be solved analytically, as the magnetic field throughout the experiment is non-constant: because the magnetic field determines the Lorentz force and varies from point to point, we are forced to solve the problem numerically using, e.g. the Euler method or, more commonly, a Runge–Kutta or Runge–Kutta–Fehlberg method [149].

The numerical integration of the trajectory of a particle presents a computational challenge in two distinct ways. Firstly, such integration is an inherently iterative process with an unknown number of steps. The number of steps is also hard to predict as it depends on both the distance to the next spacepoint (which is unknown) as well as the degree to which the magnetic field is inhomogeneous. Runge–Kutta–Fehlberg methods provide an estimate of the error incurred at each step, and may choose to increase or decrease the step size accordingly [150]. Notably, this embeds an iterative process with an unknown number of steps – namely, the propagation – inside of another iterative process which also has an unknown number of steps, namely the Kálmán filter. The second challenge is that the numerical integration process must fetch the vector representing the magnetic field at each step; in fact, it may need to do so four or more times for each Runge–Kutta step, and it may need to do this for thousands of tracks in parallel. It is imperative, therefore, that the magnetic field is stored in an efficient way that maximises the

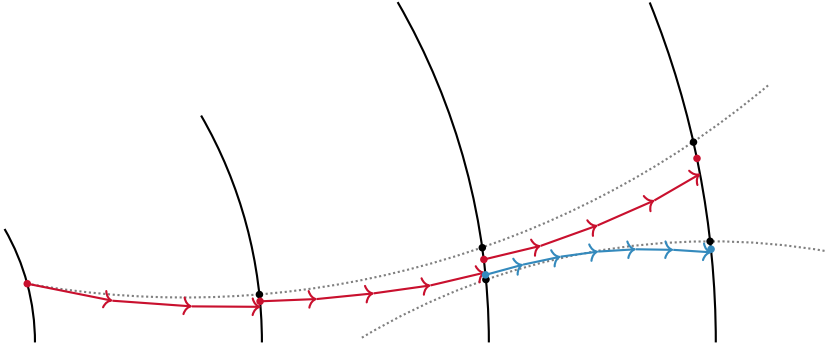


Figure 4.9: An example of combinatorial Kálmán filtering extending Figure 4.7 with an additional track. For every measurement on each surface, a new branch is started if the measurement is within the error cone of the track. In this example, two tracks – red and blue – are created. Trajectories are exaggerated for illustrative purposes; in particular, this example would require the charge of the hypothesised particle to be flipped, which is unusual in practice.

effectiveness of the caches in our computational devices.

We now conclude the description of how Kálmán filtering is used to find tracks, but this does not yet fully describe the problem at hand. Indeed, the problem described in this section is *combinatorial* Kálmán filtering, and it is brought forth by the fact that after a given time step, there may be more than one spacepoint which can extend the track. If this is the case, the Kálmán gain for each of these measurements will differ, as will the updated state estimate. In order to ensure that the largest number of high-quality tracks can be found in such cases, the Kálmán filtering has to branch, evaluating the filtering process – including the gain and update steps – separately. An example of combinatorial Kálmán filtering is shown in Figure 4.9. It may in some cases be possible to cull branches which leads to a combinatorial branch-and-bound process. The branching factor of this process is determined by the number of spacepoints in the detector and the uncertainty in the measurements.

On a final note, there are two principal strategies for combinatorial Kálmán filtering which differ in representation of the spacepoints that they select. The first strategy, which we will refer to as the *free* combinatorial Kálmán filtering, propagates tracks until a spacepoint is found in the uncertainty region around the track. This approach is conceptually simple and matches the approach hitherto described. Although much effort has been expended to implement free combinatorial

Kálmán filtering – primarily by Klimpel [151] – it involves a very large amount of intersection checks between the uncertainty volume and the spacepoints. In order to alleviate this problem, most track reconstruction software packages employ a so-called *bound* combinatorial Kálmán filtering. The bound CKF exploits the fact that spacepoints definitively lie on a set of known, flat surfaces and works by checking for intersections between the uncertainty region and those surfaces, which is significantly cheaper in computational terms. When a bound CKF intersects a surface it will retrieve a list of measurements associated with that surface and check whether those measurements lie within the uncertainty bounds or, more accurately, the projection thereof on the surface. Although the bound CKF is both conceptually and mathematically more complex, it requires far fewer clock cycles to run and it is therefore the more wide-spread approach.

The combinatorial Kálmán filtering process exhibits three computation dwarves. Firstly, it exhibits similarities to the ‘Branch-and-Bound’ pattern due to the combinatorial nature of the algorithm: branches are created when multiple spacepoints are found in a single filtering step, and these branches can be culled if they are deemed infeasible. Secondly, algorithm exhibits the ‘Dense Linear Algebra’ dwarf due to the large amount of matrix computation required to update the state in both the Kálmán steps as well as the Runge–Kutta integration. Finally, this part of the algorithm chain strongly matches the ‘Structured Grids’ dwarf as accesses to structured magnetic field data is a performance-critical part of the code. Combinatorial Kálmán filtering is additionally challenging due to the three layers of unknown imbalance that it exhibits: we branch an unknown number of times after each Kálmán step, we execute an unknown number of Kálmán steps per branch, and we need to perform an unknown number of numerical integration steps in each Kálmán step.

4.5 Track Refinement

Once the track finding stage has been completed, we are left with knowledge about which measurement belongs to which tracks. Furthermore, we have rough estimates of how the corresponding particle behaved along its track: did it scatter while traversing the detector material, or did it lose any energy in the magnetic field? These parameters are crucial to understanding the exact behaviour of the particle, and the track refinement stage serves to more accurately compute the parameters of the track throughout the detector.

4.5.1 Global and Local Fitting

The trajectory of a particle is not just described by the discrete measurements associated with it, but also by a variety of unknown variables such as the scattering angle on each of the surfaces (including non-sensitive surfaces), the energy loss along the trajectory, and the momentum of the particle. The track fitting step serves to fit values to these variables in such a way that it minimises the χ^2 value of the track. In other words, it aims to find the most likely values for these variables given the a priori knowledge we have of the track. It is worth noting that track fitting also aims to find the most likely position of the points where the particle crossed the sensitive surfaces in the detector, refining the estimates made of these positions by the combinatorial Kálmán filter, which are themselves refinements of the measurements created in the preprocessing stage.

Many different strategies for track fitting exist. These can be roughly categorised as local strategies which aim to make locally optimal fitting decisions, and global strategies which aim to simultaneously optimise parameters across the entire track. Although other algorithms in the track reconstruction chain are also dependent on the detector geometry and design to a certain extent, the track fitting step is uniquely problem-specific: different fitters do better in certain geometries and even for certain kinds of particles. Popular track fitting strategies include Kálmán filtering, the Gaussian sum filter, and the global χ^2 fitter. In this context, Kálmán filtering is very similar to the non-combinatorial methods described in Section 4.4.3 – non-combinatorial because the measurements have already been found, thus there is no need to branch – where the Kálmán updating step incorporates knowledge about measurements further along the track, knowledge which is not available during the execution of the combinatorial Kálmán filtering and which allows for higher-precision fitting. The Gaussian sum filter is a non-linear generalisation of the Kálmán filter which is able to more accurately fit non-Gaussian processes by approximating these processes as Gaussian mixtures [152]. Although Gaussian sum filtering is more computationally expensive, it is particularly suited for the reconstruction of electron tracks [153], as electrons suffer highly non-Gaussian disturbances along their trajectories. Both the Kálmán filter and the Gaussian sum filter are local methods. The global χ^2 fitter is – as the name suggests – a global fitting method. It operates by iteratively minimising the χ^2 value between the all of the track parameters and the expected parameter as given by physical models of processes such as multiple scattering [154]. Due to its importance to the track reconstruction in the ATLAS experiment, efforts have been made to optimise the

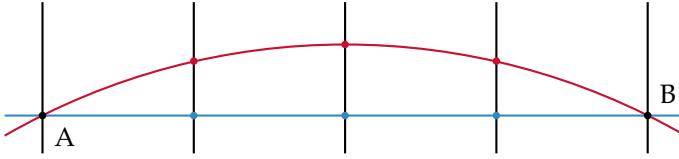


Figure 4.10: An example of ambiguity in which two tracks – red and blue – both pass through measurements *A* and *B*. To resolve this ambiguity, we need to either remove one of the tracks, or assert that measurements *A* and *B* may have been created by multiple particles.

global χ^2 fitter for CPU architectures [155].

As track fitting algorithms are diverse and problem-specific, it is particularly difficult to classify them and to develop general implementations of them. Due to the simple one-to-one relationship between the inputs and the outputs of track fitting, we find that it matches the ‘MapReduce’ dwarf. Furthermore, each of the algorithms described above makes extensive use of dense linear algebra to propagate particle states and, during these propagations, the algorithms make extensive use of magnetic field data. As such, we posit that track fitting also exhibits similarities to the ‘Dense Linear Algebra’ and ‘Structured Grid’ dwarves. Difficulties in implementing track fitting on massively parallel environments include the fact that the fitting strategy must be carefully tweaked for different purposes, which could lead to either branch divergence or the launch of very small kernels.

4.5.2 Ambiguity Resolution

A guiding assumption throughout the track reconstruction process is that each measurement and each corresponding spacepoint belongs to a single particle. Indeed, even in dense environments it is exceedingly unlikely for two particles to cross a sensitive surface such that the measurements cannot be easily distinguished. Therefore, any situation in which a single measurement is part of more than a single particle track indicates that a so-called *fake* track was formed: a track which does not correspond with a true particle. Such fakes increase the type II error rate which is undesirable. As such, most track reconstruction algorithms finish with a so-called *ambiguity resolution* algorithm which resolves the ambiguous nature of measurements being shared between tracks.

Although the assumption that each measurement corresponds to a single track is a useful one, it does somewhat reduce the efficacy of the track reconstruction in terms of efficiency, i.e. the number of tracks that is correctly reconstructed. Indeed,

in unlike cases where two (or more) tracks really did cross a sensitive surface in very close proximity, naive ambiguity resolution would remove one of these so-called *true* tracks. In order to prevent this from happening, we recall that we briefly discussed four strategies for dealing with such ambiguities in Section 4.3.1. One approach is to naively adhere to the one-to-one correspondence between particles and measurements, which would lower the recall rate for events in which measurements are shared between tracks. Another approach is to naively assume that one measurement can be shared between arbitrarily many tracks, which would significantly increase the type II error rate. These two approaches, while simple, have undesirable effects on the efficacy of track reconstruction. A third approach is to split clusters during the preprocessing stage, but this risks increasing the combinatorial explosion in processes such as seed finding and the combinatorial Kálmán filter.

The final and most common approach is to split clusters only after the track finding. Indeed, after running the combinatorial Kálmán filter, we know which measurements belong to which tracks, and after the track fitting we have access to the χ^2 value of those tracks which indicate their quality. The most common ambiguity resolution algorithms, then, proceed as follows. For each measurement, we count the number of tracks; if there is only one track, no ambiguity exists. If there are multiple tracks, we investigate the cluster from which the measurement was created, as the pattern in the activation of the cells may reveal the total number of particles involved. In the ATLAS experiment, a small neural network is able to accurately tell whether a cluster belongs to one, two, or three particles [143]. We then compare our estimate of the number of particles involved in the creation of a measurement with the number of tracks. If the number of tracks is smaller than or equal to the particle count hypothesis, the ambiguity is resolved. If it is greater, tracks must be removed until the state is no longer ambiguous. Track removal can be performed using either a local method which makes greedy, locally optimal decisions, or a global system which aims to incorporate information about the entire event. It has been shown that both approaches achieve high efficacy, although global approaches may be slightly more effective [156].

Ambiguity resolution is, especially in a global configuration, an ‘N-Body’ problem owing to the fact that there is great interaction between all tracks that were found in the event. In massively parallel environments, we predict that the primary challenge in implementing ambiguity resolution will be to expose sufficient parallelism in the solution. Indeed, local ambiguity resolution algorithms are strongly sequential in nature as tracks must be removed iteratively. Furthermore, both local

Table 4.1: A description of the different data types that are inputs, outputs, or intermediate results of the track reconstruction pipeline.

Name	Abbr.	Description
Hit	HIT	Amount of charge deposited in a single cell of a pixel-like detector.
Cluster	CLS	Cluster of hits that form an 8-neighbourhood on a single surface.
Measurement	MSM	Weighted centroid of a cluster of hits giving the passage of a particle through a surface.
Spacepoint	SPT	Three-dimensional point corresponding to a measurement on a surface.
Triplet seed	TRP	Combination of three spacepoints that describe a potential track.
Prototrack	PTK	The physical parameters of a hypothetical particle that moves along a seed.
Track	TRK	A collection of arbitrarily many spacepoints belonging to one particle, with parameters.

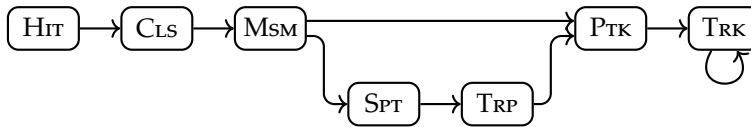


Figure 4.11: A task graph representation of the basic track reconstruction pipeline, using abbreviated data types as described in Table 4.1.

and global ambiguity resolution algorithms must communicate efficiently due to the fact that whenever a track is deleted, the state of all other measurements sharing that track must also be updated.

4.6 Summary

In this chapter, we have explored the state of the art in charged particle track reconstruction in order to answer Research Question 1. We have explored the various algorithms that are employed this task, and we have evaluated how they fit into massively parallel systems. We briefly review the different data types involved in the algorithm chain in Table 4.1, and we review the algorithms that operate on those types in Table 4.2 and Figure 4.11. In real-world track reconstruction

Table 4.2: A description of the different algorithms that constitute the track reconstruction pipeline. The ratio of each algorithm describes roughly whether the number of outputs is greater than, equal to, or smaller than the number of inputs.

Name	In	Out	Ratio	Dwarves ¹	Static
Clustering	HIT	CLS	$N : 1$	SG, SL, GT	–
Measurement creation	CLS	MSM	$1 : 1$	MR	–
Spacepoint formation	MSM	SPT	$1 : 1$	MR	DD
Seed finding	SPT	TRP	$1 : N$	NB, DP, BB	MF ³
Track parameter est.	TRP	PTK	$1 : 1$	MR	MF ³
Comb. Kálmán filter	PTK, MSM	TRK	$1 : N$	BB, DL, SG	DD, MF
Track fitting	TRK	TRK	$1 : 1$	MR, DL, SG	DD, MF
Ambiguity resolution	TRK	TRK	$N : 1$	NB	–

¹ Dwarves as classified by Asanović et al. [21], where SG is ‘Structured Grids’, SL is ‘Sparse Linear Algebra’, GT is ‘Graph Traversal’, MR is ‘MapReduce’, NB is ‘N-Body’, DP is ‘Dynamic Programming’, BB is ‘Branch-and-Bound’, and DL is ‘Dense Linear Algebra’.

² Static data requirements; ‘DD’ is the detector description, ‘MF’ is the magnetic field in the detector.

³ Although seed finding and track parameter estimation depend on the magnetic field in a detector, these algorithms usually use homogeneous approximations of it.

applications, it is not uncommon to find additional algorithms which provide both functional and extra-functional properties to the process. Non-functional enhancements might, for example, come in the form of additional culling of seeds or tracks and functional enhancements might come in the reconstruction of particle tracks that are otherwise out of reach of standard reconstruction software. One such example is the reconstruction of tracks with a large distance from the beam pipe which may originate from particle decays that happen outside of the luminous region; this process is (somewhat confusingly) referred to as *large-radius tracking*.

The functional and extra-functional parts of track reconstruction are involved in a constant dance; reconstructing more complex tracks produces potentially exciting physics results which would otherwise not be seen but, at the same time, consumes many additional clock cycles. Extra-functional enhancements to the track reconstruction chain, in turn, clock cycles and thereby allow more complex processing to happen. Although we have not discussed these additional steps here,

we will discuss some extra-functional algorithms in later chapters. The primary purpose of this thesis, however, is to reduce the computational cost of solving the problems highlighted in this chapter by offloading them to massively parallel devices.

In this process, each step of the track reconstruction chain will present its own challenges. Indeed, the different steps that constitute a track reconstruction algorithm exhibit very different computational properties: out of the 13 dwarves of parallel computation posited by Asanović et al. [21], track reconstruction resembles at least eight. There are two topics, however, which are both recurrent in the problem under study and which are also prominent in other fields of high-performance computing: performance-critical multi-dimensional data storage and imbalanced workloads between threads. In the coming chapters, we will tackle these problems in a generalised way, aiming to development methods which are abstract enough that they can be employed not only in the field of high-energy physics but more broadly across all domains of computational science.

5

Exploring the Design Space of Vector Field Representations

Composition is the way to control complexity.

— **Brian Beckman**
(Astrophysicist)

In Chapter 4, we have explored the steps involved in the reconstruction of charged particle tracks, and we have discussed some state-of-the-art implementations. Now, we proceed by examining some of the key challenges in the implementation of these algorithms on massively parallel architectures. We devise ways in which these challenges can be overcome not only in the context of track reconstruction, but also in parallel computing at large. First, we identify that track reconstruction algorithms rely on large multi-dimensional arrays which are used to represent magnetic fields. Such multi-dimensional arrays can pose performance challenges due to cache effects in high-energy physics but also in, say, computational fluid dynamics [157] and it is thus worthwhile to explore the different ways in which such data can be represented. This poses a problem however: the design space for array storage can be quite large and can impact both the functional as well as the extra-functional properties of the program. Furthermore, exploring even a single point in the aforementioned design space can be a time-consuming effort as it requires significant development efforts. In this chapter, we propose a methodology for decomposing the design space of multi-dimensional array storage so that it can be more efficiently explored in order to answer Research Question 2.

This chapter is based on the following publication:

- Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger and Attila Krasznahorkay. ‘Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Com-

position’. In: *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. ICPE’23. Coimbra, Portugal: Association for Computing Machinery, 2023, pp. 55–66. ISBN: 9798400700682. DOI: 10.1145/3578244.3583723

5.1 Introduction

Vector fields are ubiquitous in a variety of domains sciences such as meteorology [158], oceanography [159], and high-energy physics [160]. When developing applications which rely on vector fields, finding efficient data structures for storing and methods for accessing such fields can be paramount to achieving high performance. Unfortunately, there is no universal solution – let alone a performant one – for representing vector fields in software: the design space is far too large and the requirements are far too varied. In terms of functional requirements [161], for example, some applications might require two-dimensional fields while others might require three-dimensional data. Non-functionally, applications may exhibit different access patterns which can significantly affect the performance of a given implementation. Finally, the landscape of hardware on which domain applications are executed has become more complicated than ever: traditional homogeneous computing systems now compete with heterogeneous systems equipped with a variety of accelerators [162]. Thus, domain scientists must find methods of storing and accessing vector fields in heterogeneous environments which guarantee high performance in specific applications.

Currently, selecting representations of vector fields is an ad-hoc process based on developer experience and trial-and-error, neither of which provides any guarantees in finding the best-performing solutions. As far as we are aware, there are no comprehensive benchmark suites that can be used to systematically quantify and rank the performance of different vector field representations for a given application. In this chapter, we introduce a systematic benchmarking approach that aims to cover the design space of vector field representations, to expose performance-relevant elements of this space, and to be easily extendable. To do so, we explicitly decompose the aforementioned design space into *access patterns* which model a field’s *usage*, and *storage backends* which model the field’s *implementation*. We then use compile-time meta-programming to generate benchmarks across the entire design space with far less effort than would be required by a conventional trial-and-error approach. Finally, we enable developers to directly apply the results of our benchmark suite through a novel library which exposes the same domain decomposition

used by our suite for use in domain applications.

Our current design space covers five families of access patterns – each of which can be extensively configured – and nineteen components for constructing storage backends, which can be composed arbitrarily. We support benchmarks for both C++-compatible CPU platforms as well as CUDA-based General-Purpose Graphics Processing Unit (GPGPU) platforms, and we make it easy for users to add new access patterns and storage backends – including for new platforms – if the existing benchmarking suite does not suit their needs.

In short, this chapter makes the following contributions:

- We decompose domain applications of vector fields into access patterns and storage backends, thus constructing a vector field representation design space based on these two dimensions (Sections 5.3 to 5.5);
- We propose a framework for the automated generation of a large number of benchmarks using the aforementioned decomposition, thus facilitating the exploration of the design space (Section 5.6);
- We enable users to leverage our benchmark suite in real-world applications by presenting a novel library that corresponds directly to the implementation of our benchmarks (Section 5.7).

5.2 Background

In a formal mathematical sense, a vector field is a vector-valued mapping \vec{f} that assigns to every element of some set S a vector in a vector space F , such that $\vec{f} : S \rightarrow F$ [163]. For the purposes of this chapter, we have opted to restrict the codomains of our vector fields to coordinate spaces, which consist of coordinates of arbitrary dimensionality over a given algebraic field. This restriction imparts additional structure upon our vector fields such that they map more naturally onto the design of modern computer systems, and while it excludes more exotic varieties of vector spaces such as function spaces, it still allows us to model many of the vector fields that are encountered in scientific computing.

5.2.1 Related Work

The representation of vector fields – and multi-dimensional data in a more general sense – has been the subject of intense study for many years. For example, Thiyyalingam, Beckmann and Kelly [164] investigate the performance of different

layout schemes for data, but their work is limited to two-dimensional data and CPU-based platforms. Nocentino and Rhodes [165] evaluate the performance of Morton curve layouts on GPU platforms but they, too, consider only two-dimensional data and study a single application. Chatterjee, Lebeck, Patnala and Thottethodi [166] study the performance of data layout schemes in depth, but their analysis is restricted to the application of matrix multiplication. Sarawagi and Stonebraker [167] evaluate different storage methods for large amounts of scientific data, potentially including vector fields, but their analysis is tailored specifically towards secondary and tertiary storage devices, while our work focuses on in-memory arrays. Edwards and Sunderland [168] introduce *Kokkos*, a library which supports the storage of data in heterogeneous environments using a variety of representations; while this is a very versatile and useful method for storing multi-dimensional arrays, we are not aware of any functionality in Kokkos that allows users to evaluate the performance of different representations. In short, the storage of vector fields – and multi-dimensional data in a more general sense – has been the subject of intense study for many years, especially in heterogeneous environments.

In this chapter, we do not propose new access patterns or storage methods for vector fields, instead, we focus on defining and systematically exploring a *design space* for the representation of vector fields that is as large as it is precisely due to the amount of existing research into the topic. We aim to provide developers of scientific applications with a comprehensive way of comparing all these choices and their effects on application performance.

5.2.2 Notation

Throughout this chapter, we adhere to a common system of notation. When describing the types of vector fields, we use the syntax of dependent types [169]. For example, the statement $\prod_{n:\mathbb{N}} \mathbb{R}^n \rightarrow \mathbb{R}^n$ is used to mean that such a vector field exists *for all* natural values of n . We use double bracket notation to denote inclusive integer intervals, such that $\llbracket 1, 3 \rrbracket$ is equivalent to $\{1, 2, 3\}$. The symbol \mathfrak{S}_n is used to denote the set of all permutations of $\llbracket 1, n \rrbracket$. We denote vector values and vector-valued functions using overhead arrows, such as \vec{f} . We denote the rounding of numbers to the nearest integer using $\lfloor x \rceil$, while rounding down is denoted $\lfloor x \rfloor$. We use the Iverson bracket $[p]$, which evaluates to 1 if the predicate p holds, and 0 otherwise [170]. In the context of types, we use $a + b$ to denote a sum type (a term of type a or a term of type b), and $a \times b$ to denote a product type (both a term of type a as well as a term of type b); we also use a^n to denote an

n -tuple of type a .

5.3 Design Space Exploration

Our goal is to select the most appropriate representation for a given vector field. To this end, we need an approach which combines comprehensive coverage of the design options (i.e. the *design space*) with a systematic *exploration* of that space through benchmarking. In this section, we describe and motivate our envisioned design space, as well as our method for exploring it through the automated generation of benchmarks.

5.3.1 Design Space Dimensions

The performance of software using vector fields is an inextricable combination of the field’s *application*, which determines how it is *used* (i.e. what the access pattern is) and its *implementation*, which determines how it is *built* (i.e. what computation is required to produce a result). Indeed, different implementations of vector fields – even if they produce the same result – can provide wildly different performance for the same application, and it is not necessarily obvious which implementations will provide the best performance in real-world scenarios. Thankfully, the fact that application and implementation are so intertwined when it comes to performance leads us naturally to a two-dimensional decomposition of the design space for such programs into what we will refer to more specifically as *access patterns* and *storage backends*.

In this framework of thinking, an *access pattern* is an abstract model of a real-world application. Access patterns impose functional requirements on storage backends, such as the dimensionality of their vectors, and determine the locality of reference – both spatial and temporal – of vectors retrieved from a given field. In addition, access patterns are bound to specific programming platforms and, as a result, require storage backends to be compatible with a given platform. Finally, access patterns can model any additional computation that may be encountered in real-world applications, such that the performance of a given storage backend can be evaluated in context, which may impact performance due to – for example – super-scalar execution. We detail the access patterns supported by our benchmark suite in Section 5.4.

A *storage backend*, then, fulfils the functional requirements imposed by a given access pattern, and introduces additional non-functional properties; in particular,

storage backends model how much performance is lost by adding functionality – such as, for example, the interpolation of vectors – to a program. In sampled vector fields, storage backends also map indices in a high-level coordinate space onto the memory of the system; a well-chosen storage backend should be capable of translating locality of reference in the high-level coordinate space (determined by the access pattern) into locality of reference in the system’s memory, such that caches can be most effective. We explore storage backends in more depth in Section 5.5.

5.3.2 Exploration through Automated Benchmarking

Once the design space of vector field representations is defined, selecting the best performing solution for a specific application is a matter of exploring this space. Thus, we propose design space exploration through selective automated benchmarking: we define a benchmarking suite, from which we select and benchmark all representations feasible for the target application, and select the best performing one. For our benchmarking suite to be useful, we posit that it must meet three requirements. Firstly, it must be *comprehensive*: the suite must be able to approximate a large variety of real-world applications and methods of storing vector fields. Secondly, the benchmarks in our suite must be *specific*: they must be capable of identifying performance-relevant design choices and allow the user to evaluate their non-functional effects in depth and at a fine level of granularity. Finally, the suite should be *applicable*: the results of our benchmark suite should allow application developers to easily apply the results of our benchmarks to the development of their applications.

A naive solution to tackle *comprehensiveness* would be to write such a large number of benchmarks that most real-world applications would be represented by sheer chance. However, this would only shift the time-consuming exploration of the design space from the application developers to the authors of the suite. Instead, we rely on an automated exploration of the design space. In particular, we compute – at compile-time – the Cartesian product of available access patterns and storage backends, and generate a benchmark for every viable pair. This approach requires far less code to be written by hand. Additionally, our benchmarking suite is easily *extensible*: developers who find a particular application missing from the existing repository of access patterns can simply add it, and our suite will automatically generate benchmarks that combine the newly-added access pattern or application with all compatible storage backends. Correspondingly, users implementing new

Table 5.1: List of access patterns supported by our benchmarking suite.

Name	Variants	Compile-time parameters ¹
SCAN	1 CPU / 1 CUDA	$\prod_{d,d':\mathbb{N}} \prod_{S:\{\mathbb{N},\mathbb{R}\}} \prod_{T:\mathcal{V}} S^{d+d'} \rightarrow T$
RANDOM	1 CPU / 1 CUDA	$\prod_{d:\mathbb{N}} \prod_{S:\{\mathbb{N},\mathbb{R}\}} \prod_{T:\mathcal{V}} S^d \rightarrow T$
EULER [†]	2 CPU / 1 CUDA	$\prod_{d:\mathbb{N}} \mathbb{R}^d \rightarrow \mathbb{R}^d$
RK4 [†]	2 CPU / 1 CUDA	$\prod_{d:\mathbb{N}} \mathbb{R}^d \rightarrow \mathbb{R}^d$
LORENTZ [†]	4 CPU / 2 CUDA	$\mathbb{R}^3 \rightarrow \mathbb{R}^3$

[†] Access pattern is data-dependent.

¹ \mathcal{V} denotes the family of finite-dimensional coordinate spaces.

storage backends can easily benchmark their implementation against a number of existing access patterns. Our suite is implemented in C++ and makes extensive use of the Boost MP11 meta-programming library [171] to perform type-level computation. In order to register and execute our benchmarks, we rely on the GOOGLE BENCHMARK [172] framework which allows us to easily filter benchmarks at run-time and re-run benchmarks in order to gather statistically sound results.

5.4 Access Patterns

In this section, we describe the access patterns included in our benchmark suite; we present five families of access patterns, encompassing a total of sixteen variants: ten for CPU-based platforms and six for CUDA-based platforms. Most of these variants can be further distinguished through the use of compile-time parameters, which are shown in Table 5.1. For example, the EULER pattern can be compiled to operate on vector fields of any dimensionality, and it can be compiled with both single- and double-precision floating point numbers. Expanding the additional compile-time parameters of these access patterns for two- and three-dimensional cases results in a total of 208 access patterns which are distinct at compile-time. Finally, each access pattern can be configured with a series of run-time parameters which may further impact performance.

5.4.1 SCAN

The SCAN pattern – given two parameters d and d' – iterates over a d' -dimensional slice of a $(d+d')$ -dimensional vector field along each of the axes, visiting equidistant

points in lexicographic order. The remaining d dimensions are static, and the indices in these dimensions are given by a d -dimensional coordinate. It follows that setting $d = 0$ simply scans the entire vector field. The SCAN access pattern can operate on input vectors of finite dimensionality over any totally ordered monoid, but in this chapter we restrict ourselves to the real, natural, and integral numbers. The CPU-based implementation of the SCAN access pattern iterates over the axes in order. The CUDA-based implementation is slightly more complex, as it uses multi-dimensional blocks to iterate over the vector field slice. Since CUDA supports only one, two, and three-dimensional kernels [97], d' is limited to $\llbracket 1, 3 \rrbracket$ when using this access pattern on a GPU. The shape of the kernel execution blocks and grid are performance-relevant parameters, as they affect the locality of vector field accesses.

5.4.2 RANDOM

The RANDOM access pattern generates random accesses in real- or integer-valued vector fields. Given a number of points m and two coordinates describing opposite corners of a hyper-box r , we generate m uniformly random coordinates $\vec{p}_1, \dots, \vec{p}_m \in r$ and retrieve the value of the vector field at those positions. Importantly, the time taken to generate these coordinates is not taken into account when running benchmarks using this pattern; rather, the m points are generated beforehand, and retrieved from an array. The RANDOM access pattern is implemented both for CPU-based platforms as well as for CUDA-based GPUs.

5.4.3 EULER

Unlike the other access patterns mentioned so far, the EULER family of patterns introduces a dependency between the way a vector field is accessed and the contents of that vector field. Parameterised by a number of agents m and a hyper-box r , this access patterns generates agents at uniformly random initial positions $\vec{p}_1, \dots, \vec{p}_m$ in the volume r in exactly the same fashion as the RANDOM pattern. Then, a total of s steps of the Euler method [173] are used to find an approximate solution to the system of initial value problems given by the randomly generated initial positions, with the derivative function given by the vector field.

We implement three variants of the EULER access pattern. The first, EULER<DEEP>, processes agents in parallel, completing all required steps for each agent sequentially. The EULER<WIDE> method instead processes one step for each agent before repeating the process until all agents have taken all required steps; we distinguish

between these access patterns because they exhibit meaningfully different locality of reference. For GPGPU platforms, a variant of the EULER<DEEP> is available.

5.4.4 RK4

The RK4 access pattern performs – in essence – the same function as the EULER access pattern, except that it uses a fourth-order Runge–Kutta method [149] rather than an Euler method (which is, in itself, a first-order Runge–Kutta method). The difference between the Euler method and the fourth-order Runge–Kutta method is that the latter makes four sub-steps in close vicinity to each other at every step; as such, these sub-steps naturally exhibit spatial locality. The RK4 access pattern has similar variants to the EULER access pattern.

5.4.5 LORENTZ

Finally, the LORENTZ access pattern is inspired by the application of vector fields in high energy physics and is defined as numeric integration of simulated agents according to the Lorentz force. We recall the Lorentz force from Section 2.2 as the force that determines the curvature of a charged particle through a magnetic field. We recall that the force \vec{F} applied to a particle with charge q and momentum \vec{v} at position \vec{p} under the influence of two vector fields, the electric field \vec{E} and the magnetic field \vec{B} , is given as follows:

$$\vec{F}(\vec{v}, \vec{p}; \vec{E}, \vec{B}) = q(\vec{E}(\vec{p}) + \vec{v} \times \vec{B}(\vec{p})) \quad (5.1)$$

In contrast to the EULER and RK4 patterns, the LORENTZ pattern does not initialise its agents at random positions, but at a given origin \vec{o} ; in order to ensure that agents diverge, each agent is assigned a velocity vector of a given length i in a random direction. It is worth noting that the movement of agents away from the origin leads to shifts in access locality as the simulation progresses. The setup of this access pattern gives rise to an initial value problem where agents move through the vector field according to Equation 5.1, and we find approximate solutions to the final positions of the agents using both an Euler method and a fourth-order Runge–Kutta method. Since each of these methods has three distinct variants (see Sections 5.4.3 and 5.4.4), the LORENTZ pattern has a total of six variants: four variants for CPU-based platforms and two variants for GPU-based platforms. The LORENTZ access pattern is defined for endomorphic vector fields over three-dimensional real-valued coordinate spaces. Note that this access pattern is not a full-fledged physics

simulation, but rather an approximation of the access pattern such a simulation may exhibit; we simplify our access pattern by assuming identical charge and unit mass for all particles, by assuming the electric field to be zero, and by wrapping particles around when they exit the boundaries of the vector field.

5.5 Storage Backends

Where the aforementioned access patterns describe how vector fields are used, storage backends describe the inner workings of the vector fields, which impact both their functional and non-functional properties. In the context of vector fields, functional properties include – but are not limited to – the types of the input and output vectors and the way out-of-bounds accesses are sampled. Conversely, the non-functional properties of vector fields include access latency, throughput, and energy usage, which are determined by factors such as the layout of samples in memory and the use of hardware acceleration.

Even if we decompose the design space of vector field benchmarks as we have described in Section 5.3, the design space for storage backends remains dauntingly large. We posit that in order to capture this space, we need to continue our decomposition further such that we do not only describe benchmarks in terms of access patterns and storage backends, but that we additionally break the latter down into their constituent components.

5.5.1 Backend Composition

Within our benchmark suite, complex vector field implementations are constructed through the composition of simple components, of which we define two distinct classes: *primitive backends* and *transformers*. Primitive backends provide behaviour that cannot be meaningfully deconstructed into smaller components; examples of such backends include the array backend which simply looks up a vector in the one-dimensional memory of the machine, and the constant vector backend which always outputs the same vector. Primitive backends are in principle usable as stand-alone vector field backends, but they lack – by design – much of the functionality that is desired in real-world applications. This functionality is correspondingly provided by backend *transformers*. Transformers are not fully-fledged storage backends by themselves, but can be *applied* to existing backends to imbue them with additional functional and non-functional properties.

To define the structure of backend transformers, we identify three distinct re-

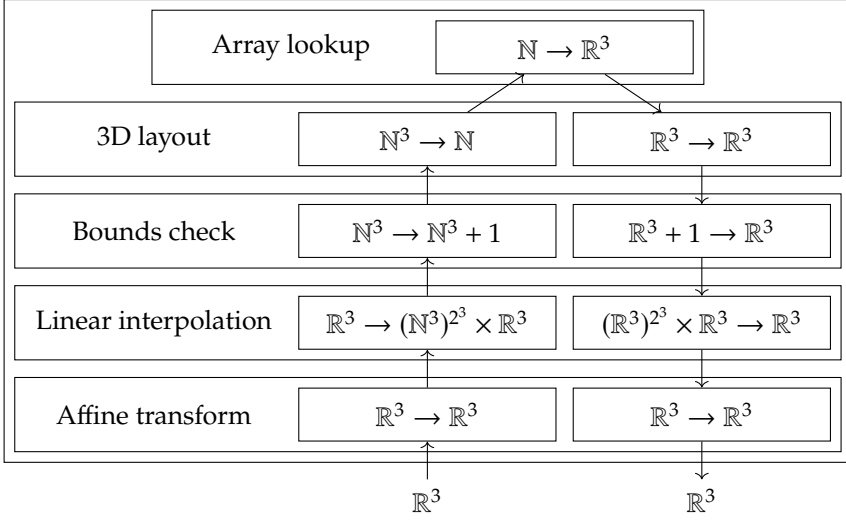


Figure 5.1: The internal structure of a storage backend, incorporating an affine transformation, trilinear interpolation, a boundary checking mechanism, and a three-dimensional layout scheme.

quirements of such transformers, the first of which is that transformers must be able to manipulate *input* coordinates before passing them to an underlying storage backend. We might consider, as an example, a backend transformer that imparts a two-dimensional layout onto an existing one-dimensional array of vectors in \mathbb{R}^2 . The function of such a transformer is to convert a coordinate of type \mathbb{N}^2 into a coordinate of type \mathbb{N}^1 . In other words, the transformer is equipped with a function of type $\mathbb{N}^2 \rightarrow \mathbb{N}^1$ which is applied to a coordinate *before* it is passed to the underlying vector field of type $\mathbb{N}^1 \rightarrow \mathbb{R}^2$, such that the entire composite backend has type $\mathbb{N}^2 \rightarrow \mathbb{R}^2$. We refer to this as the *contravariant* component of the backend transformer, in accordance with the idea that the profunctorial function arrow \rightarrow is a contravariant functor in its first argument [174]. Secondly, a backend transformer must be able to modify the *output* of its underlying backend. For instance, a transformer which discards the second component of a two-dimensional real-valued vector applies a function of type $\mathbb{R}^2 \rightarrow \mathbb{R}^1$ *after* querying the underlying storage backend. If we consider the same $\mathbb{N}^1 \rightarrow \mathbb{R}^2$ backend as before, this results in a new backend of type $\mathbb{N}^1 \rightarrow \mathbb{R}^1$. We refer to this as the *covariant* component of the transformer.

Finally, a backend transformer must be able to communicate information between its contravariant and covariant components, and it must be able to apply simple

underlying storage backends to complex structures. A prime example of these requirements is given by bilinear interpolation which, in its contravariant component, computes four integer-valued coordinates from a single real-valued coordinate as well as two weights which are used to linearly interpolate the output vector. Because the computation of the weights happens in the contravariant component of the transformer while the weighting happens in the covariant component, the transformer must somehow communicate these weights. In addition, the transformer must request *four* vectors from the underlying vector field, rather than one. Within the design we have proposed so far, both of these tasks are impossible. However, we can resolve both problems elegantly by applying a functor to the output of the contravariant component and the input of the covariant component of the transformer; in the case of bilinear interpolation, we might consider the functor $F(a) = a^4 \times \mathbb{R}^2$. This allows us to transparently lift the underlying storage backend such that communication becomes possible by embellishing our types with additional data, and we can use existing backends without the need to adapt them to more complex data types.

We are now ready to define backend transformers more formally. Given a functor F and types a_1, a_2, b_1, b_2 , a backend transformer is a term of type $(a_2 \rightarrow F(a_1)) \times (F(b_1) \rightarrow b_2)$, where a_1 and a_2 describe the contravariant component of the transformer, and the remaining types b_1 and b_2 describe the covariant component. The composition of transformers, then, is an operation of the type given in Equation 5.2:

$$\begin{aligned} \circ_T : (a_3 \rightarrow F(a_2)) \times (F(b_2) \rightarrow b_3) \rightarrow \\ (a_2 \rightarrow G(a_1)) \times (G(b_1) \rightarrow b_2) \rightarrow \\ (a_3 \rightarrow (F \circ G)(a_1)) \times ((F \circ G)(b_1) \rightarrow b_3) \end{aligned} \quad (5.2)$$

Composition is defined as in Equation 5.3:

$$(f_1, g_1) \circ_T (f_2, g_2) = (F(f_2) \circ f_1, g_1 \circ F(g_2)) \quad (5.3)$$

Furthermore, application of a transformer to a backend is an operation of the type given in Equation 5.4:

$$\$_T : (a_2 \rightarrow F(a_1)) \times (F(b_1) \rightarrow b_2) \rightarrow (a_1 \rightarrow b_1) \rightarrow (a_2 \rightarrow b_2) \quad (5.4)$$

And application is defined as in Equation 5.5:

Table 5.2: List of primitive storage backends supported by our benchmarking suite.

Name	Platform	Field type
ARRAY	CPU	$\prod_{T:\mathcal{V}} \mathbb{N} \rightarrow T$
CUDAARRAY	CUDA	$\prod_{T:\mathcal{V}} \mathbb{N} \rightarrow T$
CUDA PITCH	CUDA	$\prod_{d:\llbracket 1,3 \rrbracket} \prod_{T:\mathcal{V}} \mathbb{N}^d \rightarrow T$
CUDA TEX	CUDA	$\prod_{d:\llbracket 1,3 \rrbracket} \prod_{d':\llbracket 1,4 \rrbracket} \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$
ANALYTIC	CPU	$\prod_{S,T:\mathcal{V}} S \rightarrow T$
CONSTANT	Any	$\prod_{S,T:\mathcal{V}} S \rightarrow T$

$$(f, g) \$_T h = g \circ F(h) \circ f \quad (5.5)$$

Storage backend transformers can be applied in any order as long as their types match, and arbitrarily many transformers can be composed. In Figure 5.1, we show an example of what the internal structure of a storage backend may look like in practice. It is worth noting that transformer composition is associative, such that for any three compatible transformers t_1 , t_2 , and t_3 , $(t_3 \circ_T t_2) \circ_T t_1 = t_3 \circ_T (t_2 \circ_T t_1)$. In addition, this notion of associativity loosely applies to application under composition: for any two transformers t_1 and t_2 and a compatible vector field f , it holds that $(t_2 \circ_T t_1) \$_L f = t_2 \$_L (t_1 \$_L f)$.

5.5.2 Primitive Backends

Table 5.2 presents a summary of the primitive backends currently supported by our suite. We describe them briefly in the following paragraphs.

ARRAY

The ARRAY backend represents perhaps the most trivial in-memory vector field: a one-dimensional array of vectors. While such one-dimensional vector fields are of little use in practice, they serve as the basis for virtually all sampled vector field backends that we can construct in CPU-accessible memory.

CUDAARRAY

The CUDAARRAY backend functions similarly to the ARRAY backend, except that its contents inhabit the host-inaccessible memory of a CUDA device. Note that

this backend does *not* use the opaque arrays which NVIDIA refers to as ‘CUDA arrays’ [97]; for a backend based on these opaque structures, we provide the CUDATex backend (Section 5.5.2).

CUDAPITCH

As an alternative to ordinary CUDA device memory, we provide a primitive backend based on pitched memory allocated using the `cudaMalloc3D` API¹ [97]. The advantage of using this method is that the CUDA runtime may pad the allocation size to ensure performance-favourable data alignment. Because pitched CUDA memory opaquely handles multi-dimensional accesses, this backend primitively supports multi-dimensional coordinates.

CUDATex

CUDA textures are useful for representing vector fields [175] because they are nominally used to store texels which can be represented using one, two, three, or four-dimensional vectors of floating point numbers analogously to how pixels can be represented by grayscale values or by red, green, blue, and optionally alpha values. The advantage of using textures to represent vector fields in a broader sense is that they feature hardware-accelerated interpolation (both nearest-neighbour and linear, up to nine bits of precision [97]), various boundary checking methods, and cache-friendly storage layouts [176]. CUDA textures support real-valued inputs up to three dimensions and real-valued outputs up to four dimensions.

ANALYTIC

In some applications, vector fields can be described entirely analytically, removing the need to store the field in memory. Our benchmark suite provides a general analytic vector field, wrapping an arbitrary user-provided vector-valued function.

CONSTANT

A constant-valued vector field returns the same vector regardless of its input, performing only the bare minimum computation necessary in order to model a vector field. Such fields provide performance that is both very high and very predictable, making them useful for establishing upper bounds for the performance that can be achieved under a given access patterns.

¹This function is capable of allocating both one- and two-dimensional arrays in addition to three-dimensional ones.

Table 5.3: List of backend transformers supported by our benchmarking suite.

Name	Type
PITCHED [†]	$\prod_{n:\mathbb{N}} \prod_T \text{Type}(\mathbb{N}^n \rightarrow \mathbb{N}) \times (T \rightarrow T)$
MORTON [†]	$\prod_{n:\mathbb{N}} \prod_T \text{Type}(\mathbb{N}^n \rightarrow \mathbb{N}) \times (T \rightarrow T)$
HILBERT	$\prod_T \text{Type}(\mathbb{N}^2 \rightarrow \mathbb{N}) \times (T \rightarrow T)$
SHUFFLE	$\prod_{n:\mathbb{N}} \prod_{p:\mathfrak{S}_n} \prod_{S,T} \text{Type}(S^n \rightarrow S^n) \times (T \rightarrow T)$
DEFAULT	$\prod_{S,T} \text{Type}(S \rightarrow S + 1) \times (T + 1 \rightarrow T)$
WRAP [†]	$\prod_{S,T} \text{Type}(S \rightarrow S) \times (T \rightarrow T)$
NEAREST	$\prod_{n:\mathbb{N}} \prod_T \text{Type}(\mathbb{R}^n \rightarrow \mathbb{N}^n) \times (T \rightarrow T)$
LINEAR	$\prod_{n:\mathbb{N}} (\mathbb{R}^n \rightarrow (\mathbb{N}^n)^{2^n} \times \mathbb{R}^n) \times ((\mathbb{R}^n)^{2^n} \times \mathbb{R}^3 \rightarrow \mathbb{R}^n)$
AFFINE	$\prod_{n:\mathbb{N}} \prod_T \text{Type}(\mathbb{R}^n \rightarrow \mathbb{R}^n) \times (T \rightarrow T)$

[†] Backend transformer with additional variants.

5.5.3 Backend Transformers

The primitive backends described in Section 5.5.2 are, by design, too simple for most real-world use cases; rather, they are designed to be used in conjunction with backend transformers which imbue them with additional functional properties. Of course, most of these properties do not come for free, and virtually any behaviour we can add to a storage backend comes at the cost of performance. The advantage of our approach, even if it may seem unnecessarily granular, is that it is *specific*: it allows us to pick – and pay for – only the functionality we need. In addition, it makes it trivial to exchange transformers in order to compare their performance. In this section, we describe the different transformers that we provide as part of our benchmark in addition to a brief summary in Table 5.3, as well as the potential performance impact of those transformers. It must be noted that, although we group different backend transformers together based on their behaviour and present them in the order that they would be commonly applied to a primitive backend, our software does not categorise transformers in this way, nor does it require them to be applied in a particular order; transformers of any kind can be composed in whatever order, as long as their types line up.

CANONICAL

In most modern computer systems, memory is presented in a one-dimensional fashion; a request is made for the n -th byte in the address space, and the corresponding

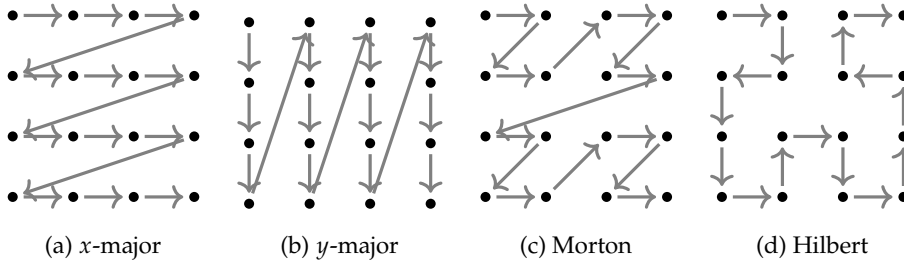


Figure 5.2: Examples of storage orders for two-dimensional arrays (of size 4×4), with arrows indicating the sequence of indices in the underlying one-dimensional memory.

byte (or, more commonly, set of bytes) is returned. In order to represent multi-dimensional arrays in a fundamentally one-dimensional address space, bijective indexing functions translate multi-dimensional coordinates into one-dimensional ones. Perhaps the most ubiquitous method for laying out multi-dimensional data is in a *canonical* fashion. Examples of two-dimensional canonical layouts in which the x and y axes are major – often referred to as the row-major and column-major layouts – are shown in Figure 5.2a and Figure 5.2b, respectively.

MORTON

Pitched storage orders provide, informally, maximal spatial locality in one dimension at the expense of locality in all other dimensions [177]. While this can be a desirable property, many real-world applications (modelled, for example, by the EULER and LORENTZ access patterns from Sections 5.4.3 and 5.4.5) exhibit locality in more than a single dimension. Such applications may benefit from laying data out according to a space-filling curve [178], which provides a compromise between locality in multiple dimensions [179]. A common example of a space-filling curve is the Morton curve, shown in Figure 5.2c. Using such a curve, a multi-dimensional index is converted to a one-dimensional index by interleaving the digits of the binary representations of the input coordinates as in the following example:

$$f(5, 3, 4) = f(\textcolor{red}{101}_2, \textcolor{blue}{011}_2, \textcolor{green}{100}_2) = \textcolor{red}{1}\textcolor{blue}{0}\textcolor{green}{1}0\textcolor{red}{1}0\textcolor{blue}{1}1\textcolor{green}{0}_2 = 342_{10} \quad (5.6)$$

The downside of Morton curve layouts is that the calculation of indices requires a significant amount of bit-manipulation, which may negate the benefits of the improved locality: d -dimensional coordinates with scalar types of width n bits

require at least $3dn$ bit-wise operations: one bit-wise disjunction, one conjunction, and one barrel shift for each bit in each dimension. It must be noted, however, that the calculation of Morton curve indices can be accelerated greatly on x86 architectures equipped with the BMI2 instruction set extension [180], which introduces the PDEP instruction². Assuming that the necessary deposition masks are computed at compile time, index calculations can be performed with as few as $2d$ operations: one bit-wise disjunction and one bit-deposition per scalar in the coordinate. Thus, indices based on Morton curves can be computed with latency and throughput similar to more canonical layouts. It is worth noting that Morton curve layouts without intermediate lookup tables require the size of the array in each dimension to be a power of two. In order to represent arrays with other sizes, the array must be padded and space must therefore be wasted. Representing a d -dimensional array in such a way requires at most $2^d - 1$ times more memory than an optimal layout. We explore the use of Morton layouts as well as a generalisation of such layouts, including their implementation, performance, and cache effects, in Chapter 6.

HILBERT

Another space filling curve is due to Hilbert [182]. Because this curve provides good locality (see Figure 5.2d for an example), its application to the storage of multi-dimensional data has been intensively studied [183, 184, 185]. The Hilbert curve is well-understood in two dimensions, but can also be generalised to three dimensions in a large number of ways [186]. Currently, we support layouts based on Hilbert curves only in two-dimensions.

SHUFFLE

All multi-dimensional layouts in our suite treat the first coordinate as the major coordinate, followed by the second, and so forth. In the two-dimensional canonical case, this is often referred to as a *row-major* layout; in order to extend this concept to an arbitrary number of dimensions, we will instead refer to it somewhat more systematically as a $(1, 2)$ -permuted layout. In a more general n -dimensional sense, all of our layouts are $(1, 2, \dots, n - 1, n)$ -permuted. Of course, such layouts may not always be optimal: in cases where multi-dimensional array accesses are anisotropically distributed along the axes, alternate permutations may provide higher performance due to caching [187].

²On most recent architectures, this instruction executes with the same latency as other bit-wise instructions, but it is emulated in microcode in pre-Zen 3 AMD processors, which may degrade performance significantly [181].

1	<code>imul 0x8(%rsi),%rdx</code>	1	<code>imul 0x8(%rsi),%rcx</code>
2	<code>add %rcx,%rdx</code>	2	<code>add %r8,%rcx</code>
3	<code>imul 0x10(%rsi),%rdx</code>	3	<code>imul 0x10(%rsi),%rcx</code>
4	<code>add %rdx,%r8</code>	4	<code>lea (%rcx,%rdx,1),%rax</code>
5	<code>mov 0x18(%rsi),%rdx</code>	5	<code>mov 0x18(%rsi),%rdx</code>
6	<code>lea (%r8,%r8,2),%rax</code>	6	<code>lea (%rax,%rax,2),%rax</code>
7	<code>lea (%rdx,%rax,8),%rax</code>	7	<code>lea (%rdx,%rax,8),%rax</code>
8	<code>vmovdqu (%rax),%xmm0</code>	8	<code>vmovdqu (%rax),%xmm0</code>
9	<code>mov 0x10(%rax),%rax</code>	9	<code>mov 0x10(%rax),%rax</code>
10	<code>vmovdqu %xmm0,(%rdi)</code>	10	<code>vmovdqu %xmm0,(%rdi)</code>
11	<code>mov %rax,0x10(%rdi)</code>	11	<code>mov %rax,0x10(%rdi)</code>
12	<code>mov %rdi,%rax</code>	12	<code>mov %rdi,%rax</code>
13	<code>ret</code>	13	<code>ret</code>

(a) Without coordinate shuffling.
(b) With (2, 3, 1)-shuffling.

Listing 5.1: Comparison of x86-64 assembly generated by gcc 11.2 with the `-O3` flag for a backend consisting of an array lookup and a three-dimensional canonical layout, with and without shuffling; although Listing (b) is generated using an additional transformer, the volume of assembly does not increase.

The number of possible permutations grows factorially with the number of dimensions; the one-dimensional case has a single permutation, the two-dimensional case has two (namely, row-major and column-major), the three-dimensional case has as many as six possible permutations, and so forth. Due to this rapid rate of growth, implementing separate storage layouts for every possible permutation is not feasible. Rather, we implement a simple family of transformers which reorder coordinates before passing them to an underlying backend. As an example, a (2, 1)-permuted shuffling transformer swaps the values of a two-dimensional coordinate, causing an underlying row-major layout to behave externally like a column-major layout. Such coordinate shuffling transformers receive their permutations as compile-time parameters, allowing them to be optimised away; a simple example of a compiler’s ability to do so is demonstrated in Listing 5.1; although the example given was compiled with the highest available optimisation flag, gcc is able to eliminate the additional shuffling code even at `-O1`; the higher optimisation level was chosen simply to reduce the size of the snippet.

DEFAULT

None of the backend transformers provided by our work incorporate boundary checking, as doing so would violate the separation of concerns between backends, and could affect performance. Of course, this means that invalid accesses invariably lead to undefined behaviour or segmentation violations. In order to alleviate this

problem, boundary checking behaviour is provided by backend transformers, including the `DEFAULT` transformer which – if the requested coordinate lies outside a given set of bounds – returns a user-provided default value.

To conclude, we will explore a possible vector field transformation in order to further elucidate our design. Consider, for the sake of example, the common case in which we wish to provide a default value – say, the zero vector – for vector field accesses which go out of bounds. We will assume that, in a general case, the type of such a transformer is $\forall n \in \mathbb{N}^+, \forall a : (a \rightarrow a + 1) \times (b + 1 \rightarrow b)$. It is worth noting that this type fits the type definition of a field transformer exactly, under the assumption that $F(x) = x + 1$; this functor models the possibility of values being non-extant³. We can equip our transformer with two points, $q, r : a^n$, which define two opposite corners of the bounding box of the vector field. The contravariant part of our transformer (with type $a \rightarrow a + 1$) then checks whether a requested coordinate lies within this bounding box; the vector is returned as normal if it is inside the bounds, and a non-extant value is returned otherwise. The underlying vector field is then mapped over this value, producing either a transformed extant value or a non-extant value, which is the argument to the covariant part of our transformer (of type $b + 1 \rightarrow b$). The covariant function then needs only check the existence of its argument, returning a zero vector if the value is non-extant. In short our transformer for out-of-bounds behaviour, t , can be defined as follows:

$$t = \left(\lambda x. \begin{cases} x & \text{if } \bigwedge_{i=0}^n \pi_i(q) \leq \pi_i(x) \leq \pi_i(r), \\ \text{Nothing} & \text{otherwise} \end{cases}, \lambda x. \begin{cases} x & \text{if } x \in a, \\ \vec{0} & \text{otherwise} \end{cases} \right) \quad (5.7)$$

The example of boundary checking serves as an excellent illustration of the power of inclusion of the functor F in the definition of our vector field transformer. Indeed, it exhibits the two main functions of the design. Firstly, it shows how functors can be used to pass information from the contravariant side to the covariant side of the field transformer without risk of this information being lost in any of the underlying transformers or, indeed, in the initial vector field to which they are applied. Secondly, this design eases the composition of transformers, as it allows us to compose a transformer which deals with non-extant values with a transformer which is completely oblivious to them; in fact, we can compose any two transformers regardless of which functor they are based on, allowing us to re-use transformers to a far greater than would otherwise be possible.

³This functor is known as `std::optional` in C++, `Maybe` in Haskell, and `std::option` in Rust.

WRAP

An alternative to the DEFAULT transformer is given by extending a vector field to infinite size, which we refer to as wrapping. We currently support clamping (extending the edges of the vector field to infinity), tiling (repeating the vector field throughout the entire space), and mirrored tiling as wrapping methods.

NEAREST

Most of the transformations hitherto described deal with integer-valued domains. While such vector fields are useful in domains such as image processing, many real-world use cases demand real-valued coordinates. In order to transform a vector field with integral-valued domain into one with a real-valued domain, we support nearest-neighbour interpolation, where the value of a real-valued coordinate is equal to the closest (by rectilinear distance) integer-valued coordinate in the underlying field [188]. This is implemented simply by point-wise rounding of the d -dimensional input vector to the nearest integer value, as follows:

$$\vec{f}(\vec{p}; \vec{g}) \approx g \left(\lfloor p_1 \rfloor \cdots \lfloor p_d \rfloor \right) \quad (5.8)$$

A notable advantage of nearest-neighbour interpolation is that it maps each real-valued coordinate onto a single integral-valued coordinate and is retractable, which allows us to write to memory that is accessed through nearest-neighbour interpolation; such an operation is not possible with linear interpolation, for example, because linear interpolation requires multiple different accesses into the underlying vector field.

LINEAR

By providing a weighted linear combination of multiple neighbouring points, linear interpolation can provide more accurate results than a nearest-neighbour method at the cost of additional computation [188]. Notably, linear interpolation in d dimensions requires 2^d accesses into the underlying vector field, all of which are guaranteed to be corners of a unit-sided hyper-cube. As a result, there exists a strong and unbiased spatial locality between the points necessary to perform such interpolation. Linear interpolation of a d -dimensional real-valued coordinate \vec{p} given an integer-indexed vector field \vec{g} is described as follows:

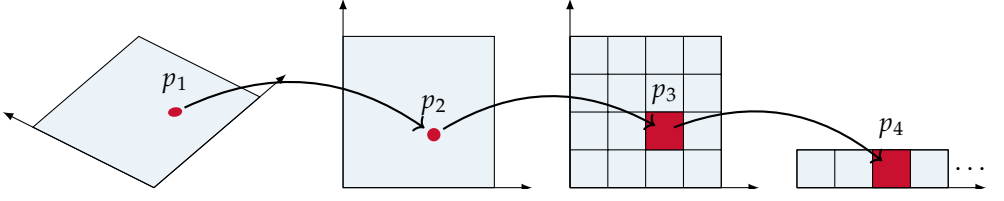


Figure 5.3: An example of how a storage backend takes a coordinate p_1 in a geometrically meaningful coordinate space to a coordinate space that maps onto a two-dimensional array (p_2 and p_3), and finally onto an address p_4 in memory.

$$\tilde{f}(\vec{p}; \vec{g}) \approx \sum_{b_1, \dots, b_d: \mathbb{B}} g \begin{pmatrix} \lfloor p_1 \rfloor + [b_1] \\ \vdots \\ \lfloor p_d \rfloor + [b_d] \end{pmatrix} \prod_{i=1}^d \begin{cases} p_i - \lfloor p_i \rfloor & \text{if } b_i \\ 1 - (p_i - \lfloor p_i \rfloor) & \text{otherwise} \end{cases} \quad (5.9)$$

AFFINE

Interpolation methods provide a way to convert index spaces addressed by positive integers to spaces that can be accessed by real numbers, but they fail to impart geometric meaning upon their input. For most real-world applications, such coordinates need to be transformed such that they map onto a more meaningful coordinate space, as shown in Figure 5.3. Currently, our benchmark suite supports arbitrary affine transformations such as translation, scaling, rotation, and shearing through a user-supplied transformation matrix. The impact of such a transformation on the performance of a vector field is potentially significant, as it requires a $(d + 1)$ -dimensional matrix-vector multiplication.

5.5.4 Performance Considerations

Although the compositional design of storage backends in our suite helps to define and explore the design space, we must ensure that this approach is not detrimental to performance; if it were, the results of our benchmarks would not be applicable to non-composite real-world vector field representations. Our approach of composing benchmarks at compile time guards us against performance degradation as a result of (1) dynamic function dispatching due to run-time polymorphism; and (2) reductions in the optimisation space afforded to the compiler.

Indeed, our approach entirely avoids the overhead of dispatch table look-ups

which are necessary when employing run-time polymorphism [189]; because the entire code path is known at compile-time, there is no need to dynamically dispatch function calls. While such look-ups are often not performance-critical in larger applications, we envision vector fields as very-high throughput structures where such overhead would be undesirable.

Secondly, a compositional approach risks reducing the optimisation space of the compiler by enabling – or, depending on the design, requiring – the compiler to emit separate symbols for all possible transformers. During the subsequent composition phase, the compiler may be unable to optimise the code *across* different components of the storage backend, thereby degrading performance. Through the use of compile-time composition, we afford the compiler a complete view of all the components of a storage backend, allowing it to optimise across function boundaries. An example of this was shown previously in Listing 5.1, where a compiler was able to completely eliminate an additional backend transformation, incorporating additional behaviour into the vector field without incurring overhead.

Thirdly, composing storage backends at compile time allows us to trivially port code to C++-based heterogeneous programming platforms such as CUDA, as the CUDA compiler can simply re-instantiate templates for compilation to PTX whenever necessary. Finally, a useful consequence of compositional software design is that it allows not only for *building* composite software, but also for the composite *reasoning* about that software. Compositional reasoning about the performance of computer systems is especially popular in the distributed systems community [190], but it has similarly been applied at the single-node scale relevant to our work [191]. By providing a compositional approach to multi-dimensional array storage, we enable the building of accurate analytical models on the performance of applications.

5.6 Evaluation

In this section, we demonstrate the use of our benchmark suite in the application under study in this thesis. In particular, we will apply it to the propagation of charged particles through magnetic fields, an important component of track reconstruction.

```

1 benchmark::register_product_bm<
2     boost::mpl1::mp_list<
3         Lorentz<Euler>,
4         Lorentz<RungeKutta4>,
5         RungeKutta4Pattern,
6         EulerPattern,
7         Random,
8         Scan>,
9     boost::mpl1::mp_list<
10         FieldConstant,
11         FieldTex<TexInterpolateLin>,
12         FieldTex<TexInterpolateNN>,
13         Field<InterpolateNN, LayoutStride>,
14         Field<InterpolateNN, LayoutMortonNaive>,
15         Field<InterpolateLin, LayoutStride>,
16         Field<InterpolateLin, LayoutMortonNaive>>>>();

```

Listing 5.2: An example of C++ code used to generate our CUDA benchmarks. Given a compile-time list of six access patterns and a list of seven storage backends, we generate forty-two benchmarks.

5.6.1 Experimental Setup

To test our benchmarks on data that is representative of real-world scenarios, we use a solenoidal magnetic field generated by the ACTS software package [122]; this vector field is representative of data used in real-world high-energy physics applications – including track reconstruction – and uniformly samples a three-dimensional magnetic field in a range of $-10\,000$ mm to $10\,000$ mm in the x - and y -axes, and a range of $-15\,000$ mm to $15\,000$ mm in the z -axis. The data is sampled at a resolution of 100 mm, resulting in a total of $201 \times 201 \times 301$ samples. We store this data using single-precision IEEE 754 floating-point numbers, resulting in a total data size of 145.9 MB.

We execute our benchmarks on six distinct devices. Five of these devices, namely an AMD EPYC 7402P CPU of the *Zen 2* microarchitecture [192, 193] as well as NVIDIA A2, RTX A4000, RTX A6000, and A100⁴ GPUs [64] of the *Ampere* architecture, are part of the DAS-6 cluster [194]. The final device, an Intel Xeon E5-2630 v3 CPU [195] of the *Haswell* microarchitecture, is part of the DAS-5 cluster [194]. The code for our CPU-based platforms was compiled with gcc 11.2, and code for the NVIDIA GPU was compiled using nvcc 11.5 (targeting PTX versions 8.0 and 8.6).

5.6.2 Benchmarking Method

The first step in applying our benchmarking-based design space exploration approach is to select an access pattern that approximates the targeted application.

⁴The PCIe model with 40 GB of HBM2e memory.

In this case, we use the LORENTZ access pattern, designed to closely represent the domain-specific application we are investigating (see Section 5.4.5). For the sake of brevity, we test the variants of this access pattern using only the Euler method – in a depth-first fashion – although our compositional approach to generating benchmarks makes it trivial to repeat this analysis for other access patterns: Listing 5.2 shows an example of how a much broader range of benchmarks can be generated using the compile-time meta-programming techniques described in Section 5.3.2.

Next, we construct a variety of storage backends with the necessary functional properties, namely (1) three-dimensional indexing; (2) real-valued indexing; and (3) appropriate transformations from the field’s real-world geometry. This leads us to the composition of fourteen storage backends; of these, seven are suited for CPU-based experiments and seven are suited for GPU-based experiments. These backends include constant-valued vector fields and various combinations of primitive backends, multi-dimensional layouts, and interpolation schemes. In our experiments, the constant-valued vector fields serve as an upper bound for the performance of a certain access pattern on a given device, capturing the maximum throughput of the computation inherent to the application rather than the access to the vector field. It is worth noting that we only investigate a subset of storage backends that could be constructed for the application under investigation; indeed, additional functional behaviour such as boundary checking could be incorporated, but we chose to omit this due to the combinatorial growth of the number of resulting experiments.

The selection of our six devices with seven storage backends each yields a collection of forty-two benchmarks. Each of these benchmarks has a number of run-time parameters, which can additionally be varied. For this case study, we set the number of steps to 8192, the step size to 10^{-3} time units, and the number of agents to 65 536. This leaves us with a final parameter, the magnitude of the initial velocity vector, which we vary between 2^8 and 2^{18} millimetres per time unit. As we grow the initial velocity of the agents in our simulation, particles take larger steps through the vector field resulting in reduced locality, and understanding the magnitude of this effect is the goal of our analysis.

With our collections of benchmarks now selected and parameterised, we obtain a total of 882 distinct experiments. For each of these experiments, we measure the throughput – in accesses per second – from the vector field by dividing the total number of accesses by the total wall-clock runtime of the benchmark. Each experiment is repeated fifty times, and we report the mean throughput of these runs. Our benchmark suite automatically gathers additional descriptive statistics

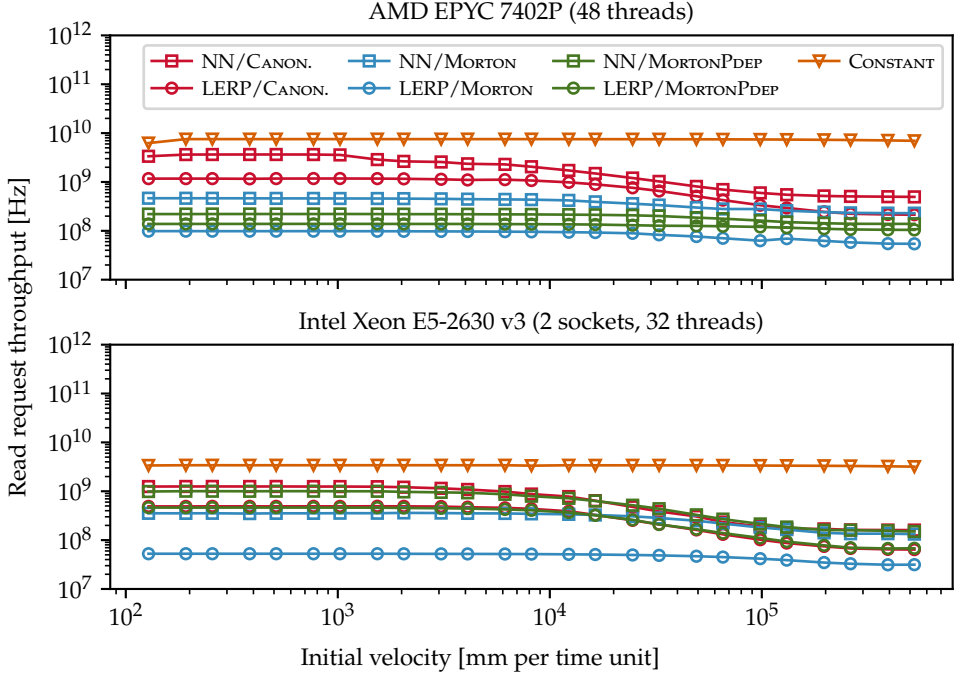


Figure 5.4: Vector field access throughput for the LORENTZ access pattern with a depth-first Euler method for two different CPUs. We use NN for nearest-neighbour interpolation, and LERP for linear interpolation.

such as the minimal and maximal throughput as well as the variance between runs, but we have omitted these metrics from this section because the variance of our results is very small.

5.6.3 Results

The results of our experiments are shown in Figure 5.4 for the CPU platforms and in Figure 5.5 for the GPU platforms. From the perspective of a developer, inspection of these data provides valuable insights that motivate application development. We analyse a few such insights in the following paragraphs.

Firstly, the representation of vector fields in these modelled applications is an important design decision: the difference in throughput between different storage backends can be an order of magnitude or larger. Furthermore, our benchmark provides a clear ranking between different backends at each point in the chosen parameter space: given a specific device and parameterisation of the access pattern,

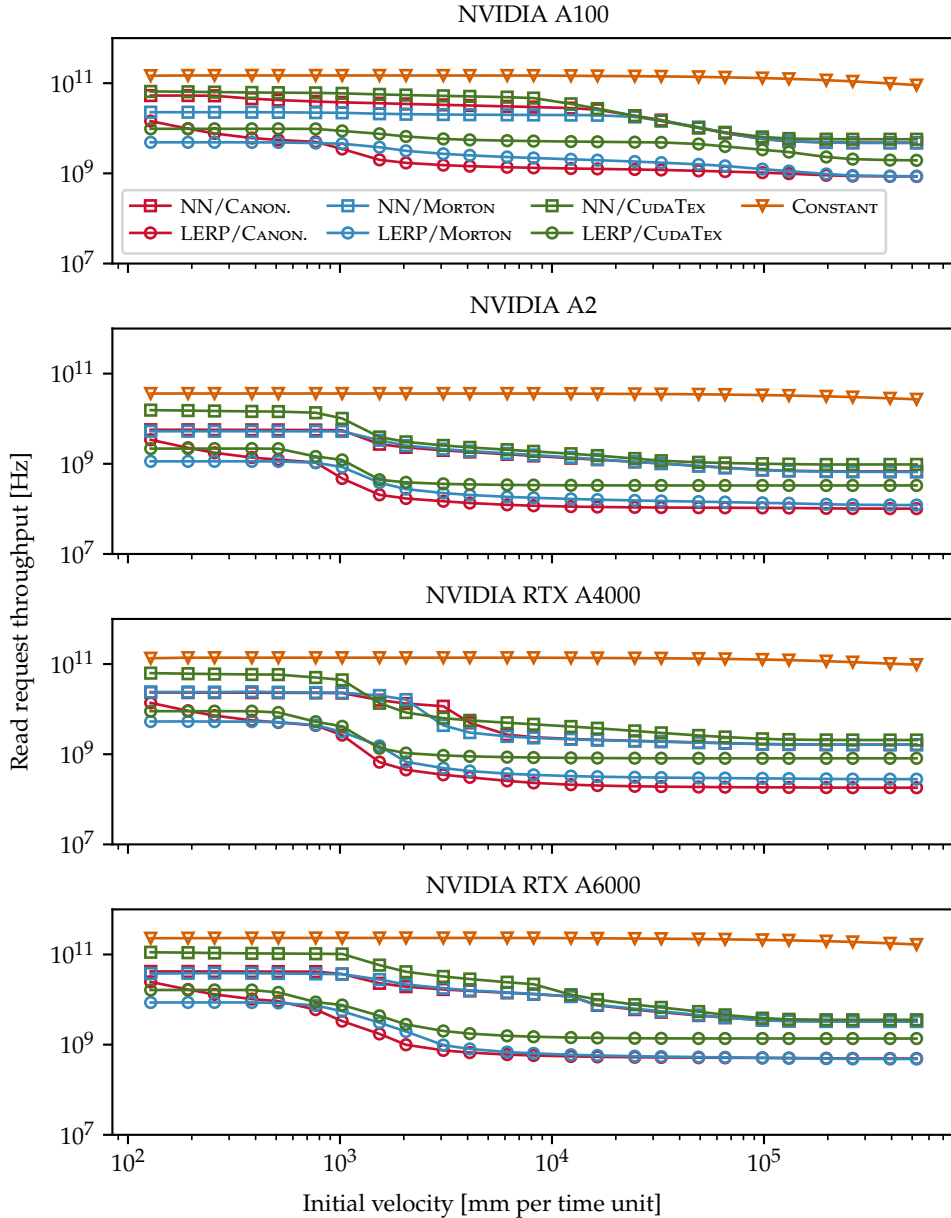


Figure 5.5: Vector field access throughput for the LORENTZ access pattern using 128 threads per block for four different NVIDIA GPUs. We use NN for nearest-neighbour interpolation, and LERP for linear interpolation.

the storage backend with the highest throughput can be easily selected. The inclusion of constant-valued vector fields provides additional insights into the upper bounds of storage backends, thus allowing us to compare other backends against this bound and calculate the fraction of maximally achievable performance obtained with each backend.

Secondly, our results show that the performance of vector fields is also strongly dependent on the properties of the access pattern. Even though all of our results are based on the LORENTZ access pattern, performance behaviour varies significantly with the initial velocity of the simulation agents, which correlates with the locality of reference in the logical three-dimensional space. Unsurprisingly, all storage backends across all devices perform worse as the initial velocity increases and locality decreases, but the degree at which performance is lost varies significantly between storage backends. This is apparent, for example, in the results for the NVIDIA A100: the storage backend incorporating linear interpolation and Morton curve indexing performs relatively poorly for lower initial velocities, but outperforms the canonical layout at higher initial velocities due to the increased preservation of locality afforded by the Morton layout. For the NVIDIA RTX A4000, we observe three distinct regimes: at low and high initial velocities, the storage backend based on texture memory outperform other backends with nearest neighbours, but at intermediate velocities the canonical layout and Morton layouts provide the best performance.

Finally, our results indicate strong sensitivity of our benchmarks to the properties and features of the hardware on which it is executed. Figure 5.5 shows the results of our benchmark when executed on four NVIDIA GPGPUs of the same architecture; beyond constant-factor performance differences due to differences in memory bandwidth and arithmetic performance, the devices under investigation exhibit different behaviour across the range of parameters tested. For example, the results for the NVIDIA RTX A6000 shows that Morton curve layouts for linearly interpreted fields outperform canonical layouts only in a specific regime of initial velocities – between approximately 1024 and 4096 millimetres per unit of time – whereas this regime extends indefinitely on the NVIDIA RTX A4000. Although we do not fully understand the source of this discrepancy, these results indicate that our benchmark can also be used to motivate the choice of processor or accelerator for an application.

Although a complete analysis of the data presented in this chapter is beyond the scope of this chapter, we find the results presented in this section to be strong indicators of the *comprehensiveness* of our benchmarking methodology: we were

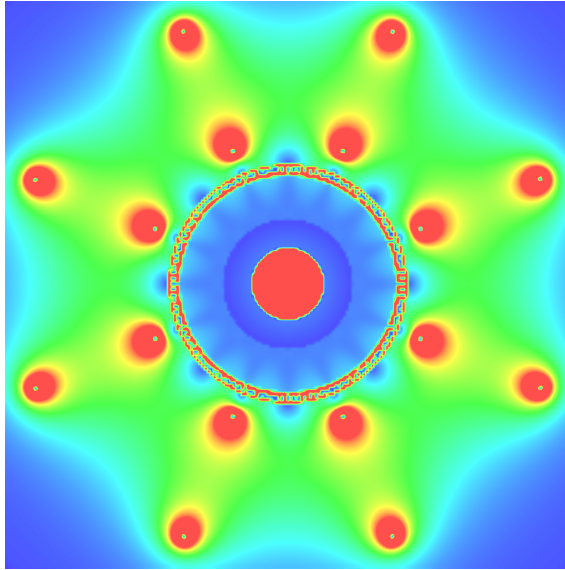


Figure 5.6: A rendering of a slice of the ATLAS magnetic field strength rendered on an NVIDIA GPU using texture memory through the library described in this chapter.

able to gather results using a variety of storage backends through little more effort than selecting the benchmarks at run-time. Furthermore, our results are *specific* as they show clear differences in performance between different storage backends: even relatively similar backend which differ by only a single component exhibit significantly performance patterns, indicating that our suite is capable of comparing different representations at fine levels of granularity.

5.7 Applicability and Limitations

The primary purpose of our benchmark-based design space exploration is to guide developers in the selection of appropriate vector fields representations for their target applications. Once the selection is made, developers are free to implement their own vector fields from scratch. However, we have already decomposed the design space of such implementations, and this decomposition may be valuable to users beyond enabling the construction of our benchmark suite: our benchmark implementation – by design – allows direct code re-use in the target application. In order to provide as much utility as possible to the developers of high-performance applications, our implementations can be re-used as a C++ library which allows

```

1 template <typename field_t>
2 __global__ void render(
3     typename field_t::view_t vf,
4     char * out,
5     unsigned int width,
6     unsigned int height
7 )
8 {
9     int x = blockDim.x * blockIdx.x + threadIdx.x;
10    int y = blockDim.y * blockIdx.y + threadIdx.y;
11
12    if (x < width && y < height) {
13        float fx = x / static_cast<float>(width);
14        float fy = y / static_cast<float>(height);
15
16        typename field_t::output_t p =
17            vf.at(fx * 20000.f - 10000.f, fy * 20000.f - 10000.f, z);
18        out[height * x + y] = static_cast<char>(std::lround(
19            255.f *
20            std::min(std::sqrt(p[0] * p[0] + p[1] * p[1] + p[2] * p[2]), 1.0f)
21        ));
22    }
23 }

```

Listing 5.3: A CUDA kernel written using our library which produces a rendering of the magnitude of a slice of a vector field, such as Figure 5.6.

users to compose vector field implementations from the same components that we use in the benchmark. In essence, our benchmark suite and library are co-designed such that the library allows for the construction of benchmarks and real-world applications, while the benchmark allows insight into the performance of the different vector fields that can be constructed using the library. Our library provides features such as loading and storing vector fields to disk, and converting vector fields between different storage backends (including between CPU- and GPU-based backends, which automatically moves memory between devices wherever necessary). An example of how our library can be used in CUDA kernels is given in Listing 5.3 and the result is shown in Figure 5.6.

We are confident that both the idea of composing *applications* and *implementations* of data structures at compile time can be generalised to other classes problems using similar techniques with the ones proposed in this chapter. Vector fields are excellent candidates for such an approach due to the wide range of access patterns and storage backends. Applications and data structures for which the design space is smaller might benefit less from the generation of benchmarks (as these benchmarks would be fewer), but the benefits of systematic exploration and automation remain relevant. Similarly, the decomposition of storage backends is transferable to other data structures, and it would be especially interesting for other kinds of multi-dimensional data. However, the dimensions of the design

space could be less broad, because functionality such as interpolation and spatial transformations do not apply to all multi-dimensional structures. In a nutshell, the generalisation of the approach to other applications is easy to achieve, but its urgency is determined by the complexity of the application domain.

Finally, it is worth noting that our benchmark suite has limitations of which users should be aware. Chiefly, our suite models applications at an abstract level that omits non-functional effects that might be present in real-world scenarios. For example, the access patterns in our suite assume that the vector field is the only data structure being used by the application at a given time. In real-world applications, multiple regions of memory may be accessed at the same time by one or more threads. This may significantly alter the cache behaviour and thereby the performance of an application when compared to our benchmarking suite. Furthermore, our suite does not currently support accessing data stored on persistent media (such as through `mmap`) which may be relevant for applications using vector fields too large to fit entirely in the main memory of a given machine. Finally, we do not currently support fields of heterogeneous vectors such as fields that combine vectors of both single and double precision floating point numbers, or vectors of different dimensionalities.

5.8 Reproducibility and Reusability

The benchmarking suite and library discussed in this chapter, the data gathered in our experiments, and the scripts used to process and visualise those data are permanently archived on Zenodo [196] and have been made available at [doi:10.5281/zenodo.7019829](https://doi.org/10.5281/zenodo.7019829). This artifact was submitted to the artifact evaluation track of the ICPE'23 conference, where it was awarded the *ACM Artifacts Available* and *ACM Artifacts Evaluated – Reusable* badges. Furthermore, at the time of writing the software – named `COVFIE`, short for *COmpositional Vector FIElds*⁵ – is available freely under the MPL-2.0 license on GitHub at <https://github.com/acts-project/covfie> [197].

5.9 Summary

Vector fields are performance-critical components of scientific applications, but we lack the tools to quantify and rank their performance. In this chapter, we introduce

⁵And, of course, an allusion to the beautiful drink that made possible this entire thesis.

a comprehensive benchmarking suite that enables developers of applications using vector fields to select appropriate, high-performance vector field representations through systematic design-space exploration. To create a comprehensive, specific, and applicable benchmark suite, we decompose vector field representations into access patterns and storage backends. These can be combined at compile-time to construct hundreds of unique benchmarks, each with additional run-time parameters. We further decompose storage backends into primitive backends and backend transformers, which can be used to add functional and non-functional properties to vector fields, both in a benchmarking setting as well as in domain-scientific use cases. We show that the use of template meta-programming allows us to automatically generate a large number of high-performance benchmarks, providing software that compromises neither performance nor usability.

In order to evaluate the efficacy of our benchmarking suite and to answer Research Question 2, we have applied it to analyse the performance of a range of vector field storage backends in an environment inspired by the combinatorial Kálmán filter as covered in Chapter 4. Thereby, we have provided evidence that the implementation of vector fields can play an important role in achieving high performance in real-world applications, and we have demonstrated that our benchmarking suite is capable of capturing the design space for such implementations.

6

Finding Cache-Friendly Data Layouts Through Evolution

*A man who dares to waste one hour of time
has not discovered the value of life.*

— **Charles Darwin**
(Natural historian)

In Chapter 5, we have explored the design space of multi-dimensional array storage. This design space consists of boundary checking methods, interpolation methods, and – importantly – array layouts. We have studied row-major layouts, column-major layouts, and the so-called *Morton* layout – the last of which is given by a particular index computation based on bit interleaving, but what happens if we change the pattern in which we interleave those bits? In this chapter, we reveal that the Morton layout can be generalised to a very large family of array layouts, each of which has its own cache properties. It is, in some sense, a design space inside a design space. We show that this design space of Morton-like array layouts can be explored efficiently using evolutionary algorithms and we demonstrate that this strategy of exploration can be used to increase the performance of arbitrary applications based on multi-dimensional arrays by significant factors in a problem-agnostic fashion. Furthermore, our method is far less invasive than traditional cache optimisation techniques and requires less programmer effort. We thereby expand on our answer to Research Question 2.

This chapter is based on the following publication:

- Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger and Attila Krasznahorkay. ‘Using Evolutionary Algorithms to Find Cache-Friendly Generalized Morton Layouts for Arrays’. In: *Proceedings of the 2024 ACM/SPEC International Conference on Performance Engineering*. ICPE’24. London, United Kingdom: Association for Computing Machinery, May 2024, pp. 83–94. doi: 10.1145/3629526.3645034

6.1 Introduction

Structured multi-dimensional data are ubiquitous in high-performance computing [198]: three-dimensional fluid simulations, dense linear algebra operations, and stencil kernels are just a few examples of applications which rely fundamentally on multi-dimensional arrays. In spite of the importance of such applications, however, most modern computer systems have one-dimensional memories: from the perspective of the programmer, memory is nothing more than a very large one-dimensional array of bytes. This discrepancy between application requirements and hardware design requires programmers to carefully consider *array layouts*: injective functions which translate multi-dimensional indices into one-dimensional memory addresses.

Although array layouts do not impact the functional properties of programs, choosing a suitable layout can significantly impact application performance in modern processors with complex cache hierarchies [199]. Exploiting these caches is of critical importance to achieving high performance in all but purely compute-bound applications, but doing so requires locality of access – both temporal and spatial – in memory. Kernels often exhibit locality in multiple dimensions, and a well-chosen array layout maximises the degree to which this application-level locality is translated to the address-level locality that caches are designed to exploit; as a result, that layout increases the efficacy of hardware caching and – by extension – the performance of an application.

Data in two-dimensions is commonly laid out in *row-major* order (shown in Figure 6.2a for an 8×8 array) or *column-major* order (Figure 6.2t) which provide good locality of access in a single dimension, but poor locality in all others. Thankfully, the design space for data orderings – in two dimensions or more – extends far beyond these canonical layouts: the *Morton* layout (Figure 6.2f), for example, is a layout based on a space-filling curve which provides balanced locality between multiple dimensions [164, 200]. Our work explores a family of data layouts which generalise the Morton order, and allow us to carefully tune the cache behaviour in multiple dimensions to match a given application.

The design space of the aforementioned family of data layouts is dauntingly large; indeed, the number of possible layout for arrays at scales applicable to real-world problems is so large that it renders exhaustive search infeasible. In order to find suitable array layouts in tractable amounts of time, we propose to employ genetic algorithms – heuristics known to be able to efficiently find high-quality solutions in large search spaces [201]. To this end, we design a chromosomal

representation of Morton-like array layouts, as well as a fitness function that uses cache simulation to estimate the performance of individual array layouts. Finally, we evaluate our evolutionary strategy and the array layouts it discovers.

In short, this chapter makes the following contributions:

- We characterise the design space given by a generalisation of the Morton array layout, and we show that the size of this design space renders exhaustive search infeasible (Section 6.3);
- We propose an evolutionary methodology based on genetic algorithms for exploring the aforementioned design space based on the simulated cache-friendliness of layouts (Section 6.4);
- We design and execute a series of experiments to assess the accuracy of our fitness function, the efficacy of our evolutionary process, and the performance of the discovered array layouts, showing that our method can improve performance up to a factor ten (Section 6.5).

6.2 Background and Related Work

In this section, we provide a brief overview of the basic concepts and notations which are essential to the remainder of this chapter, and highlight relevant related work.

6.2.1 Indexing Functions and Canonical Layouts

Dense n -dimensional arrays can be imagined as structured grids in which each element is assigned to exactly one point in \mathbb{N}^n . In most modern processors, multi-dimensional arrays are a software-level abstraction over the one-dimensional memory of the machine; in order to actually access multi-dimensional data, we need to define a function that converts indices in n dimensions to memory addresses¹. We refer to the class of such functions as *indexing functions*, and they are isomorphic to *array layouts*. In short, an n -dimensional indexing function is an injective (often bijective) function of the following type, where N_i represents the size of the array in the i th dimension, \times is the generalised Cartesian product, and $\llbracket a, b \rrbracket$ is the integer interval from a to b :

¹In reality, address calculations must also consider array offsets (the address of the first element) and scales (the size of each element). We skip over these complications as they are handled transparently by address generation units in modern hardware, and they affect all array layouts in the same manner.

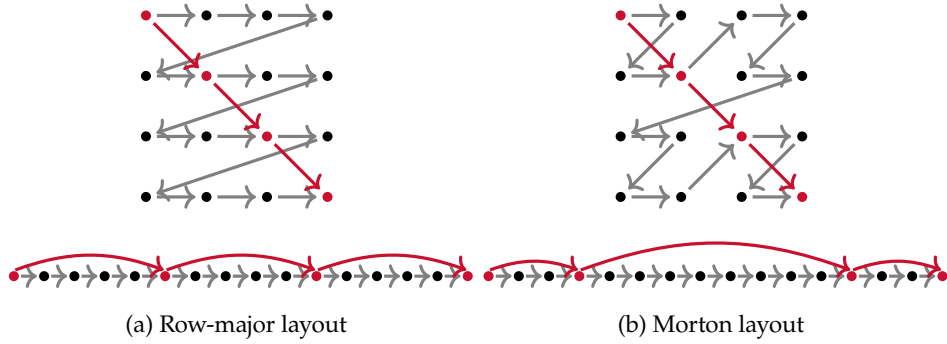


Figure 6.1: Two-dimensional arrays laid out in memory along the gray arrows. An application accesses the array diagonally along the red arrows. Application locality is shown above, memory locality is shown below.

$$f : \bigtimes_{i=0}^{n-1} \llbracket 0, N_i - 1 \rrbracket \rightarrow \left\llbracket 0, \left(\prod_{i=0}^{n-1} N_i \right) - 1 \right\rrbracket \quad (6.1)$$

In a multi-dimensional grid, we denote the elements along a given axis – that is to say, the sequence of elements for which all indices except one are fixed – as *fibres* [202]. In a two-dimensional case, fibres along the x -axis are known as *rows*, and fibres along the y -axis as *columns*. In order to facilitate the description of arrays in three or more dimensions, we use the term *mode- m* fibres to describe fibres along the m th dimension, such that mode-0 fibres are synonymous with rows, mode-1 fibres refer to columns, and so forth.

The most common group of multi-dimensional indexing functions are the *canonical* layouts, sometimes known as the *lexicographic* layouts or, in the two-dimensional case, the *row-* and *column-major* layouts. In a canonical layout, one-dimensional array indices are calculated according to Equation 6.2, in which x_0, \dots, x_{n-1} are components of the n -dimensional index, and N_0, \dots, N_{n-1} represent the size of the array in each dimension:

$$f(x_0, \dots, x_{n-1}; N_0, \dots, N_{n-1}) = \sum_{i=0}^{n-1} \left(\prod_{j=0}^{i-1} N_j \right) x_i \quad (6.2)$$

An important corollary of Equation 6.2 is that the mode-0 fibres are contiguous in memory, i.e. Equation 6.3 holds:

$$f(x_0 + 1, x_1, \dots, x_{n-1}) = f(x_0, x_1, \dots, x_{n-1}) + 1 \quad (6.3)$$

It is worth noting that the calculation of addresses in column-major layout – in which the mode-1 fibres are contiguous – is also given by Equation 6.2, with the order of the indices and sizes swapped. The canonical array layouts achieve perfect spatial locality in one dimension: if a kernel accesses memory along mode- m fibres, then a canonical layout where the m th dimension is major will provide the optimal translation between locality in the multi-dimensional space to locality in memory. Many real world applications, however, exhibit locality in multiple dimensions; a kernel might, for example, iterate diagonally over an array; an example of this – and the resulting locality in memory – is given in Figure 6.1a.

The performance of canonical storage layouts has been studied extensively. Park, Hong and Prasanna discuss methods for compensating for the weaknesses of canonical layouts using tiling and recursive layouts [199]. Similarly, Kowarschik and Weiß propose a variety of strategies that mitigate cache misses in canonical storage layouts for numerical applications [203]. Weinberg, McCracken, Strohmaier and Snively propose a metric for the locality of array layouts [204]. Jang, Schaa, Mistry and Kaeli analyse the performance of access patterns in multi-dimensional data in graphics processing units (GPUs) [205]. Che, Sheaffer and Skadron propose a method for automatically optimising storage layouts [206].

6.2.2 Morton Layouts

The Morton order is a notable example of a non-canonical array layout that provides balanced locality in multiple dimensions. It is conceptually simple to understand, efficient to implement in commodity hardware (as we will show in Section 6.3.3), and it has been shown to positively affect the efficacy of hardware caches: Al-Kharusi and Walker show the efficacy of the Morton layout in molecular dynamics applications [207], Perdacher, Plant and Böhm describe its benefits in matrix decomposition [208], and Thiyagalingam, Beckmann and Kelly provide an in-depth performance analysis of this array layout in a range of kernels [164]. Chatterjee, Lebeck, Patnala and Thottethodi show the applicability of Morton layouts – as well as other non-canonical layouts – in matrix multiplication [209], and their work is expanded upon in [210]. Applications of the Morton order in more than two dimensions have been studied by Pawłowski, Uçar and Yzelman [211]. Mellor-Crummey, Whalley and Kennedy show the applicability of array layouts based on space-filling curves – like the Morton layout – for irregular applications [212].

The practical applicability of the Morton layout is further evidenced by the OPIE compiler, which employs Morton array layouts natively [213].

The performance benefits of the Morton layout stem from its spatial structure: an example – which justifies why this layout is sometimes known as the *Z-order layout* – is given in Figure 6.1b; note the difference in locality in the address space compared to the canonical layout (Figure 6.1a). The Morton order layout has also been applied to data movement in parallel systems by Walker and Skjellum [214], and Deford and Kalyanaraman have applied the layout to workload distribution in parallel processes [215]. Bader explores a variety of applications of space-filling curves in scientific programs [216]. Armbrust et al. explore the application of Morton curves for the storage of databases, reducing the total amount of data read from persistent storage [217]; although the aforementioned paper considers a much higher level of abstraction than the methods in this chapter – which operate at the level of hardware caches – we believe that the methods presented in this chapter may generalise to a broader range of applications, including databases.

In the Morton order, multi-dimensional indices can be converted to one-dimensional addresses in a variety of ways. The Moser–de Bruijn sequence [218] is commonly used as it allows efficient conversions in two dimensions, but this method requires us to store the Moser–de Bruijn sequence in memory, and accessing this sequence causes additional overhead. Therefore, we prefer a different method based on the interleaving of the (unsigned) binary representation of multi-dimensional indices. As an example, the two-dimensional index $(3, 5)$ can be bijectively mapped into one-dimensional memory by finding the binary expansions of the indices, i.e. $(011_2, 101_2)$, and interleaving the bits yielding $100111_2 = 39_{10}$. This is equivalent to first dilating and shifting the binary expansions of the numbers, and then taking their bitwise disjunction (OR): the first index is dilated yielding 000101_2 while the second index is dilated and shifted left yielding 100010_2 . Taking the bitwise disjunction of these numbers yields the same address as using the interleaving strategy. The computation of Morton indices through bit manipulation can be extended to an arbitrary number of dimensions; the three-dimensional index $(3, 5, 4)$ expands to $(011_2, 101_2, 100_2)$, and the resulting memory address is $110001011_2 = 395_{10}$. Note that the relative significance of bits in each of the input indices is preserved in the output address. Gottschling, Wise and Adams present the idea that the Morton layout can be generalised by allowing arbitrary bit-interleaving orders [219, 220], which is foundational to our work. This idea is further expanded on by Walker [221].

6.2.3 Genetic Algorithms

Genetic algorithms are a class of heuristics introduced by Holland which are designed to solve optimisation and search problems by emulating the process of evolution as it happens in the natural world [222]. In genetic algorithms, *generations of individuals*, i.e. sets of candidate solutions, iteratively explore a design space through genetic operators. In particular, *crossover* operators model the combination of the genetic material of two or more individuals [223], and *mutation* operators model random changes to the gene pool [224]. In genetic algorithms, individuals are removed from the population based on their *fitness*, i.e. the quality of the solution they represent to the problem posed [225]. Genetic algorithms have seen successful application in an extremely broad range of fields, ranging from drug discovery [226] to music composition [227]. Genetic algorithms have also proven useful for design space exploration in computer systems; Pimentel shows that they can be applied in the design of embedded systems [228]. Sapra and Pimentel show that a broader class of evolutionary approaches can be used in the design of neural networks [229]. The optimisation problem we consider in this chapter is combinatorial in nature, and the application of genetic algorithms to such problems has also been extensively studied and proven across a variety of domains [230, 231, 232, 233]

6.3 Generalised Morton Layouts

The Morton layout functions by interleaving the bits of the input indices in a fixed pattern: bits are drawn from each of the inputs in a round-robin manner. In this section, we generalise this idea, allowing bits to be interleaved in arbitrary order. This gives rise to more specialised layouts with different structure and, as a result, different extra-functional properties [219, 220, 221]. Figure 6.2 shows all 20 layouts that are given by different bit interleaving orders for an 8×8 array. As with the standard Morton layout, the generalised Morton layout can be applied to any number of dimensions. As an example, the following three-dimensional layout selects two bits from the second index, one bit from the third index, then two bits from the first index, etc.:

$$f(011_2, 101_2, 100_2) = \begin{array}{r} 000011000_2 \\ 000100001_2 \\ \vee 100000000_2 \\ \hline 100111001_2 \end{array} = 313_{10} \quad (6.4)$$



Figure 6.2: All 20 layouts for 8×8 arrays generated by the family of indexing schemes described in Section 6.3. Note that Figure 6.2a corresponds to a row-major layout, while Figure 6.2t corresponds to a column-major layout.

Our goal is to find Morton-like layouts, i.e. bit-interleaving patterns, that improve application performance through an increase in cache efficacy. In this section, we will show that the design space for such layouts is very large, motivating the use of genetic algorithms. This necessitates a chromosomal representation of layouts, which we also present in this section. In addition, we describe how the canonical layouts can be described using the same representation, and we delve into practical considerations such as the computational cost of computing indices and support for Single Instruction Multiple Data (SIMD) processing.

6.3.1 Enumerating Layouts

We can characterise Morton-like layouts by the bit scattering pattern applied to each of the inputs (e.g. for Equation 6.4, the first index is scattered to the fourth, fifth, and eighth bits). However, such a characterisation is unsound in the sense that it allows us to describe invalid layouts: if two bits from any of the input indices are mapped onto the same bit in the output, the bitwise disjunction becomes an information-destroying operation and the layout becomes non-injective – that is, it would cause multiple multi-dimensional indices to map onto the same location in memory, making the layout unusable.

We can instead characterise layouts in a manner that is both complete and sound by enumerating the *source* of each bit in the output index. In the remainder of this chapter we shall denote array layouts using sequences of the form $[i_0, \dots, i_{n-1}]$, indicating the source indices in order of increasing bit significance: the least significant bit in the output index is drawn from the i_0 th input index, the second-least significant bit is drawn from the i_1 th input, and the most significant bit is drawn from the i_{n-1} th input. Note that each input bit must be used once and only once: whenever a bit is to be drawn from a given input index, we implicitly use the least significant bit for that input which has not yet been consumed. For the layout shown in Equation 6.4, the two least significant bits are drawn from the second input, the third-least significant bit is drawn from the third input, and so forth: the resulting array layout is denoted using the sequence $[1, 1, 2, 0, 0, 1, 2, 0, 2]$.

The aforementioned characterisation of multi-dimensional layouts gives rise to families of layouts. The family of layouts over n inputs, where each input has b_0, \dots, b_{n-1} bits, is isomorphic to the set of permutations of the multiset $S = \{0 : b_0, \dots, n-1 : b_{n-1}\}$. We denote this set of permutations as $\mathfrak{S}(S)$. For convenience, we obviate the intermediate multiset such that $\mathfrak{S}'(b_0, \dots, b_{n-1}) = \mathfrak{S}(\{0 : b_0, \dots, n-1 : b_{n-1}\})$. We can then determine the total number of possible layouts as the

number of multiset permutations of \mathfrak{S}' [234, p. 42]:

$$|\mathfrak{S}'(b_0, \dots, b_{n-1})| = \binom{\sum_{i=0}^{n-1} b_i}{b_0, \dots, b_{n-1}} = \frac{(\sum_{i=0}^{n-1} b_i)!}{\prod_{i=0}^{n-1} (b_i!)} \quad (6.5)$$

6.3.2 Including Canonical Layouts

It is worth noting that canonical layouts over arrays for which the size in each dimension is a power of two are, in fact, members of the family of Morton-like layouts. In order to sketch an informal argument for this, we recall that the indexing function for an n -dimensional canonical layout given array sizes N_0, \dots, N_{n-1} is defined as in Equation 6.2. If we assume that all sizes are powers of two, then the product of these sizes is guaranteed to be itself a power of two. Because multiplication by powers of two can be interpreted as a left-ward shift, the canonical layouts shift each input index x_0, \dots, x_n to a specific location in the binary expansion of the output index. Furthermore, because we assume $\forall i : x_i < N_i$, each bit in the output is determined by exactly one of the input indices; this allows us to interpret the summation as a series of bit-wise disjunctions, exactly like the definition of our Morton-like layouts. In general, a mode-0-major canonical layout of a $2^{b_0} \times \dots \times 2^{b_{n-1}}$ array can be characterised – in the the scheme defined in Section 6.3.1 – by contiguous subsequences of bits, each drawn from the same index, i.e. a sequence of the following form:

$$\underbrace{[0, \dots, 0]}_{b_0 \text{ times}} \underbrace{[1, \dots, 1]}_{b_1 \text{ times}} \dots \underbrace{[n-1, \dots, n-1]}_{b_{n-1} \text{ times}} \quad (6.6)$$

Canonical layouts with different major axes can be constructed by changing the order of the contiguous subsequences. The fact that the canonical layouts are members of the Morton-like family of array layouts allows us to evaluate the performance of these layouts in the exact same framework as the rest of the Morton-like layouts, and we will exploit this in Section 6.5.

6.3.3 Hardware-Accelerated Indexing

It is tempting to extend the aforementioned ideas to even more exotic indexing functions, like the Hilbert array layout [182, 235, 236]. The computational cost of many such functions renders them impractical, however: if the cost of computing memory addresses is too large, any performance gained by improving the cache-friendliness of a program will be negated. The Morton-like layouts we consider in

this chapter allow efficient index calculations on modern commodity hardware, which we demonstrate in this section.

Under canonical array layouts, indices are calculated either iteratively through repeated addition and multiplication, or in parallel through parallel multiplication followed by reduction through addition. In n -dimensional cases both approaches require $n - 1$ additions and $n - 1$ multiplications, operations which can be efficiently performed on virtually all processors. Specifically, the Intel Haswell and AMD Zen 3 microarchitectures – on which we focus in this chapter – can perform 64-bit register addition (`ADD r64 r64`) with a latency of 1 cycle and a reciprocal throughput of 0.25 cycles, while they can execute multiplication (`IMUL r64 r64`) with a latency of 3 cycles and a reciprocal throughput of 1 cycle [237].

Our bit-interleaving array layouts rely, in n -dimensional cases, on $n - 1$ bitwise disjunctions and n bit-scatter operations. Such disjunctions (`OR r64 r64`) can be performed with a latency of 1 cycle and a reciprocal throughput of 0.25 cycles – the same as the `ADD` instruction – on both of the aforementioned microarchitectures. We perform the bit-scattering operation using the *parallel bit deposition* (`PDEP r64 r64`) instruction, which is included in the BMI2 extension to the x86-64 instruction set [80]. The Intel Haswell and AMD Zen 3 microarchitectures both perform bit deposition with a latency of 3 cycles and a reciprocal throughput of 1 cycle, identical to the `IMUL` instruction. It follows that Morton-like indexing requires – in theory – only a single additional instruction over canonical index calculation.

The hardware extension required to perform bit deposition is widely supported: BMI2 has been included in Intel processors starting with the Haswell microarchitecture (2013) [238], and in AMD processors starting with the Excavator microarchitecture (2015), albeit in a limited fashion; AMD processors gained full hardware support for these instructions starting with the Zen 3 microarchitecture (2020) [239]². Compiler support for the BMI2 instruction set extension is also widely available; an example of how easily Morton-like layouts can be implemented in software is given in Listing 6.1.

In order to further evaluate the competitiveness of Morton-like layouts compared to canonical layouts, we analyse implementations of both indexing schemes over a range of dimensionalities as compiled by GCC 12.3 and CLANG 15.0 using OSACA 0.5.2 [240]. All code was compiled using the `-O2` optimisation flag. The results of this analysis are shown in Figure 6.3. Over the range of dimensionalities considered, the canonical layouts are consistently faster, i.e. require fewer cycles to

²Pre-Zen 3 processors supported parts of the BMI2 instruction set – the `PEXT` and `PDEP` instructions in particular – through emulation in microcode rather than in hardware, making them very slow.

<pre> 1 #include <x86intrin.h> 2 3 template <std::size_t N, std::size_t... J> 4 std::size_t morton_idx_helper(5 const std::array<std::size_t, N> &i, 6 const std::array<std::size_t, N> &m, 7 std::index_sequence<J...>) 8 { 9 return (_pdep_u64(i[J], m[J]) ...); 10 } 11 12 template <std::size_t N> 13 std::size_t morton_idx(14 const std::array<std::size_t, N> i, 15 const std::array<std::size_t, N> m) 16 { 17 return morton_idx_helper(18 i, m, std::make_index_sequence<N>() 19); 20 } </pre>	<pre> 1 ; 1 dimension 2 pdep %rsi,%rdi,%rax 3 ret 4 5 ; 2 dimensions 6 pdep %rdx,%rdi,%rax 7 pdep %rcx,%rsi,%rsi 8 or %rsi,%rax 9 ret 10 11 ; 3 dimensions 12 mov 0x8(%rsp),%rax 13 mov 0x10(%rsp),%rcx 14 mov 0x18(%rsp),%rdx 15 pdep 0x20(%rsp),%rax,%rax 16 pdep 0x28(%rsp),%rcx,%rcx 17 or %rcx,%rax 18 pdep 0x30(%rsp),%rdx,%rdx 19 or %rdx,%rax 20 ret </pre>
---	--

(a) Templated C++ code which scales to arbitrarily many dimensions. (b) x86-64 assembly code generated by GCC 12.3.

Listing 6.1: An example implementation of Morton layouts with arbitrary bit-interleaving masks implemented in C++, as well as the generated x86-64 assembly.

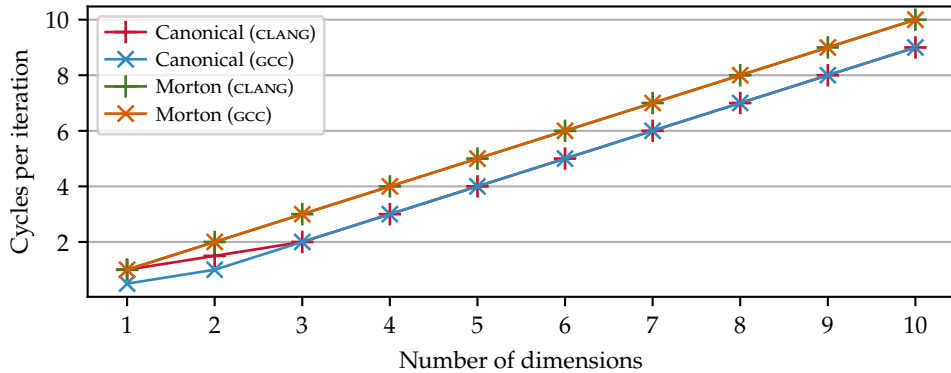


Figure 6.3: Throughput of a kernel calculating array indices using canonical layouts as well as Morton-like layouts on the Intel Haswell microarchitecture as given by OSACA.

compute, than the Morton-like layouts. However, the difference in performance – approximately one cycle – is relatively small and overshadowed by the number of cycles saved due to a reduction in cache misses. Furthermore, we focus primarily on memory-bound applications, in which a small increase in index calculation time is unlikely to affect performance. We conclude, therefore, that Morton-like layouts are competitive with canonical layouts strictly in terms of address computation costs.

6.3.4 Support for SIMD

An important consideration in the design of array layouts is the ability to vectorize kernels through Single Instruction Multiple Data (SIMD) operations. Canonical layouts guarantee the contiguity of fibres in the array, which facilitates the (automated) vectorisation (e.g. the application of SIMD) of many operations, and this benefit is lost when applying the array layouts discussed in this chapter. However, we posit that there remains ample opportunity to accelerate computation on Morton-like arrays using SIMD, and we argue this by distinguishing two classes of computation patterns.

The first class consists of *unstructured* patterns in which data is operated on element-wise without spatial context, i.e. without consideration of nearby elements; a prominent example of such an operation is matrix *addition*. In such applications, SIMD can be trivially applied to the underlying one-dimensional memory, regardless of the layout of the data: since elements can be added point-wise in any order, doing so in the order in which the data is laid out in memory is both feasible and enables SIMD.

The second class of problems consists of *structured* patterns in which operations must be performed in a specific order. A prime example of such an operation is matrix *multiplication* where the inner product of fibres must be computed. In such cases, it is imperative that fibres can be accessed in contiguous blocks. The size of these blocks depends on the vectorisation technology used as well as the size of the data type: in the x86 instruction set, SSE vectorisation requires two consecutive double-precision numbers or four consecutive single-precision numbers [241]; the much wider ARM SVE instruction set extension [81] may require up to thirty-two consecutive double-precision numbers or sixty-four single-precision numbers.

In order to facilitate vectorisation for structured patterns of computation, we can impose certain constraints on the array layouts we consider. Indeed, if the n least-significant bits of an interleaving pattern are all drawn from the m th input index,

then the layout guarantees that the mode- m fibres in the array are contiguous in blocks of 2^m elements. This requirement can be incorporated into the selection of array layouts; for example, we can enable efficient AVX2 vectorisation (with a vector width of 256 bits) using single-precision (32-bit) floating point numbers by ensuring that the three least significant bits in an array layout are drawn from the same source. In other words, we can easily constrain our search space to include only array layouts with properties that favour vectorisation, and we believe that doing so will enable SIMD-accelerated computation arrays laid out in Morton-like orders.

6.4 Exploration Through Evolution

The canonical set of indexing bijections for laying out multi-dimensional memory is small: for two-dimensional data, there are two possible layouts, and the performance of these layouts can be evaluated using exhaustive benchmarks [1, 164, 242]. Exhaustively exploring the family of indexing function outlined in Section 6.3, however, is impractical owing to the sheer number of permissible permutations. Importantly, the number of canonical layouts increases only with the number of *dimensions*, while the number of Morton-like layouts increases with both the number of dimensions and the *size* of the array in each of those dimensions. By Equation 6.5, a small 4×4 array (indexed by two bits in each dimension) can be laid out in $(2+2)!/2!2! = 6$ ways. A larger array of size 4096×4096 (twelve bits in each dimension) can be laid out in $(12+12)!/12!12! = 2\,704\,156$ ways. A three-dimensional array of size $256 \times 256 \times 256$ has the same number of elements as the aforementioned 4096×4096 array, but permits $(8+8+8)!/8!8!8! = 9\,465\,511\,770$ permutations. As these examples indicate, the number of possible permutations quickly scales beyond what can be feasibly explored through exhaustive search; in order to tackle the explosive growth in the design space for Morton-like layouts, we propose the use of genetic algorithms (Section 6.2.3).

6.4.1 Genetic Algorithm Configuration

In this chapter, we employ a relatively simple (λ, μ) -ES genetic algorithm [222, 243]. The chromosomal representations of array layouts is identical to the characterisation given in Section 6.3.1, and this gives rise to a combinatorial optimisation problem. We facilitate the recombination of array layouts into novel layouts using the Ordered Crossover (OX) operator [244], and we employ inversion-based

mutation [245]. Our approach differs from classical genetic algorithms in only one significant way: our initial population is not chosen randomly from the solution space. Instead, the initial populations for our evolutionary experiments always consist of two individuals, depicting two canonical layouts for a given array size, as described in Section 6.3.2. We choose to do this to ensure that our initial populations are unbiased and deterministic, allowing us to more easily assess the efficacy of our genetic strategy.

6.4.2 Fitness Function Design

There are two general strategies for evaluating the performance, i.e. the fitness, of a given array layout under a given cache hierarchy and access pattern: measurement and simulation. In order to assess fitness through *measurement*, we execute a program on actual hardware and measuring the running time of the process. Although such a fitness function is conceptually simple, it suffers from two primary flaws: (1) measurements are noisy and may suffer from run-to-run variance, which may hinder the performance of genetic algorithms [246] – in particular, our genetic algorithm is vulnerable to noise stemming from cache pollution effects; and (2) measurements require access to the target hardware, which may be inconvenient or even impossible – for example, in hardware-software co-design scenarios, where the hardware under consideration does not (yet) exist. For these reasons, we choose not to base our fitness function on measurements.

Instead, we employ *simulation* for which we need a simulator that can accurately compare the cache performance for different access-patterns on the same cache hierarchy. For this, we selected PYCACHESIM, a component of the KERNCRAFT toolkit [247]. We use PYCACHESIM by simulating an access pattern such as matrix multiplication and registering the relevant trace of load and store operations. After all accesses have been recorded, we force a write-back of the caches and collect the number of hits and misses in each cache level. We combine the number of hits in every cache level as well as in main memory with the latency of retrieving data from each of these levels to compute the total number of cycles spent retrieving data from the cache hierarchy. Given an array layout I , an access pattern A and a simulated cache hierarchy H , we calculate the total number of cycles using the following equation, in which Li_{hit} , Li_{miss} , and Li_{lat} represent the number of hits, the number of misses, and the latency of the i th cache level³, and M represents main memory:

³In this chapter, we follow the ubiquitous convention that the cache closest to the core is named L1, but our software supports cache hierarchies of arbitrary depths and with arbitrary naming schemes.

$$C(I; A, H) = M_{\text{hit}}(I; A, H)M_{\text{lat}}(H) + \sum_i L_{i\text{hit}}(I; A, H)L_{i\text{lat}}(H) \quad (6.7)$$

From this, we compute an approximation of the number of accesses performed per cycle, giving rise to a higher-is-better fitness function defined as follows:

$$F(I; A, H) = \frac{L1_{\text{hit}}(I; A, H) + L1_{\text{miss}}(I; A, H)}{L1_{\text{lat}}(H) \cdot C(I; A, H)} \quad (6.8)$$

Intuitively, the numerator in Equation 6.8 counts the total number of memory accesses, as all accesses either hit or miss in L1. The denominator, then, estimates the total number of cycles spent retrieving data from the various cache levels. The denominator is multiplied by a normalising factor equal to the latency of the L1 cache; it follows from Equation 6.7 that the achievable performance is softly bound by the reciprocal of the L1 access latency. Indeed, this performance is achieved if and only if all accesses hit the L1 cache. Normalising the fitness function using the L1 cache latency improves our ability to compare results between different cache hierarchies.

6.5 Evaluation

We evaluate the efficacy of the methods hitherto discussed by demonstrating that (1) our fitness function is well-chosen, i.e. that it correlates with performance measurements in real hardware; that (2) our evolutionary process is capable of finding novel array layouts with favourable cache properties; and that (3) the layouts which are found by our evolutionary process actually lead to relevant performance gains in real hardware. Our validation is based on eight distinct access patterns and two processors with distinct cache hierarchies.

6.5.1 Experimental Setup

We consider a set of eight access patterns loosely based on the selection of algorithms used by Thiyaalingam, Beckmann and Kelly [164]. The access patterns were picked to represent common real-world applications (dense linear algebra and fluid dynamics), to represent both two-dimensional and three-dimensional applications, and to differ in critical properties such as memory size and number of loads and stores. A description of the access patterns we consider in this chapter is given in Table 6.1.

```

1 template <concepts::array<2> M>
2 void mm_ijk(const M & A, const M & B, M & C) {
3     const auto m = C.get_size();
4     for (std::size_t i = 0; i < m; ++i) {
5         for (std::size_t j = 0; j < m; ++j) {
6             typename M::value_type acc = 0.;
7             for (std::size_t k = 0; k < m; ++k)
8                 acc += A.load(i, k) * B.load(k, j);
9             C.store(acc, i, j);
10        }
11    }
12 }

```

Listing 6.2: Example of how an access pattern (MMijk) is described in C++. Meta-programming allows the same source to be used for both simulation and execution on real hardware.

All our access patterns are described using C++ code – see the example in Listing 6.2 – which ensures high performance as opposed to the Python code used for our evolutionary processes; the interaction between the C++ and Python components of our project is managed using PYBIND11 [248]. We use template meta-programming to generalise our access patterns in such a way that a single definition can be used for both simulation and benchmarking without loss of performance due to run-time polymorphism; this eliminates any possible discrepancies between the code used for simulation and the code used for measurement.

We conduct our experiments on two different CPUs: the Intel Xeon E5-2660 v3 [250] based on the Haswell microarchitecture [238], and the AMD EPYC 7413 [251] based on the Zen 3 microarchitecture [239]. When we perform experiments on non-simulated Haswell processors we use the the DAS-6 cluster [194], whereas we use a machine located at CERN for experiments on Zen 3 processors. When we perform experiments based on simulation, we use the the DAS-6 cluster [194] and configure our cache simulator according the cache configurations shown in Listing 6.3a for the Haswell processor, and Listing 6.3b for the Zen 3 processor. Note that the cache configurations are based on the accessibility of caches from a single core. This is especially relevant for the L3 cache on the Zen 3 chip, which is shared across groups of cores rather than the entire CPU: in the case of the AMD EPYC 7413, the CPU comes equipped with 128 MiB of L3 cache, but only 32 MiB is accessible from any single core [239]. We simplify the cache replacement policies of the actual hardware by assuming LRU caches (i.e. caches with a least-recently-used eviction policy); in reality, the Haswell caches employ eviction policies consistent with tree-based pseudo-LRU for the L1 and L2 caches [237, 252], while the L3 cache is consistent with a set-dueling-controlled adaptive insertion

Table 6.1: Overview of the access patterns used for evaluation, including the use of memory and the number of loads and stores.

Access pattern	Description	Mem. size	Loads	Stores
MMTJrk($m; s$)	Multiplication of two $2^m \times 2^m$ matrices, both of s -byte real numbers.	$3 \cdot s \cdot 2^{2m}$ B	$2 \cdot 2^{3m}$	2^{2m}
MMTJrk($m, n; s$)	Multiplication of a $2^m \times 2^n$ matrix by a transposed $2^m \times 2^n$ matrix.	$s \cdot (2 \cdot 2^{m+n} + 2^{2n})$ B	$2 \cdot 2^{2m+n}$	2^{2m}
MMMkj($m; s$)	Same as MMTJrk($m; s$) with the order of the inner loops switched.	$3 \cdot s \cdot 2^{2m}$ B	$3 \cdot 2^{3m}$	2^{3m}
MMTJrk($m, n; s$)	Same as MMTJrk($m, n; s$) with the order of the inner loops switched.	$s \cdot (2 \cdot 2^{m+n} + 2^{2n})$ B	$3 \cdot 2^{2m+n}$	2^{2m+n}
JACOBI2D($m, n; s$)	Four-point stencil kernel over a $2^m \times 2^n$ array of s -byte real numbers.	$2 \cdot s \cdot 2^{m+n}$ B	$\sim 4 \cdot 2^{m+n}$	2^{m+n}
CHOLESKY($m; s$)	Cholesky–Banachiewicz decomposition of a $2^m \times 2^m$ matrix.	$2 \cdot s \cdot 2^{2m}$ B	$2 \cdot 2^{2m}$	$\sim \frac{1}{2} \cdot 2^{2m}$
CRout($m; s$)	Crout decomposition of a $2^m \times 2^m$ matrix of s -byte real numbers.	$2 \cdot s \cdot 2^{2m}$ B	$\frac{7}{2} \cdot 2^{2m}$	2^{2m}
HIMENO($m, n, p; s$)	Nineteen-point Himeno stencil [249] over $2^m \times 2^n \times 2^p$ arrays.	$12 \cdot s \cdot 2^{m+n+p}$ B	$24 \cdot 2^{m+n+p}$	2^{m+n+p}

<pre> 1 caches: 2 L1: 3 sets: 64 4 ways: 8 5 line: 64 6 replacement: LRU 7 write_back: true 8 store_to: L2 9 load_from: L2 10 latency: 4 11 L2: 12 sets: 512 13 ways: 8 14 line: 64 15 replacement: LRU 16 write_back: true 17 store_to: L3 18 load_from: L3 19 victim_to: L3 20 latency: 12 21 L3: 22 sets: 25600 23 ways: 16 24 line: 64 25 replacement: LRU 26 write_back: true 27 latency: 36 28 memory: 29 first: L1 30 last: L3 31 latency: 200 </pre>	<pre> 1 caches: 2 L1: 3 sets: 64 4 ways: 8 5 line: 64 6 replacement: LRU 7 write_back: true 8 store_to: L2 9 load_from: L2 10 latency: 7 11 L2: 12 sets: 1024 13 ways: 8 14 line: 64 15 replacement: LRU 16 write_back: true 17 store_to: L3 18 load_from: L3 19 victim_to: L3 20 latency: 12 21 L3: 22 sets: 32768 23 ways: 16 24 line: 64 25 replacement: LRU 26 write_back: true 27 latency: 46 28 memory: 29 first: L1 30 last: L3 31 latency: 200 </pre>
(a) Intel Xeon E5-2660 v3	(b) AMD EPYC 7413

Listing 6.3: Two examples of cache specifications for different CPU models. Note that these configurations are approximations of the true cache hierarchies.

policy [237, 253]. Cache sizes were gathered from specification documents [238, 254], while cache latencies were obtained optimistically from sources on the fastest load-to-use latencies [254, 255]. The Zen 3 L1 cache has a fastest load-to-use latency of four cycles for integers and seven cycles for floating point values [254] – we use the latter in our simulations. Finally, we assume a constant 200 cycle access latency for main memory in both systems.

6.5.2 Fitness Function Validation

The fitness function we use in our evolutionary process (Section 6.4.2) is based on simulation results because simulation yields significant benefits over empirical measurements, primarily in terms of determinism and in the ability to simulate future hardware. However, this strategy is not without risk: the simulation we perform is based on a non-cycle-accurate simulator, uses simplified cache hierarchies,

and ignores computation entirely. Consequently, we must evaluate the usefulness of our fitness function by establishing its correlation with execution time in real hardware.

Ideally, the running time of a kernel using a given array layout would correlate inversely linearly with our fitness function, therefore ensuring two important properties. Firstly and most importantly, it guarantees that running time decreases monotonically with the value of the fitness function, such that an array layout with a higher fitness value is guaranteed to run more quickly; this allows us to establish a ranking of layouts and enables us to reliably select the best-performing array layout. Secondly, linear correlation guarantees proportionality between fitness and running time, which facilitates the weighted selection of individuals.

To evaluate the degree to which the aforementioned criteria are met, we randomly select one hundred array layouts for each of the eight access patterns given in Table 6.1. We then evaluate the simulated fitness and measure the running time in real hardware of each pair of array layout and access pattern. The fitness functions of the pairs are calculated in parallel, as they are designed to be deterministic and impervious to cache pollution or resource contention. The empirical benchmarks are performed sequentially, ensuring that the benchmark is the sole user of the processor caches. All measurements are repeated ten times, and we report the mean and standard deviation of the running time.

The results of this experiment are shown in Figure 6.4. The coefficient of variation of the measurements never exceeded a value of $c_v = 0.0801$. Accordingly, we have opted to omit error bars from the figure. Upon visual inspection, it is clear that the correlation between our fitness function and running time is not linear, although the two do appear correlated. We confirm our suspicions of correlation by computing Pearson's coefficient of correlation (ρ_p) and Spearman's coefficient of rank correlation (ρ_s); the resulting statistics are given in Table 6.2. We observe that our fitness function and running time correlate moderately to strongly with running time for the Intel Xeon E5-2660 v3 processor, although the correlation is weaker for the AMD EPYC 7413 processor. Although it is clear that there is space for the fitness function to be improved, we believe that it correlates sufficiently with running time to enable its use in genetic algorithms.

6.5.3 Genetic Algorithm Performance

To evaluate our evolutionary process (Section 6.4) as a whole, we intend to verify that it can, indeed, find Morton-like array layouts that have a higher simulated

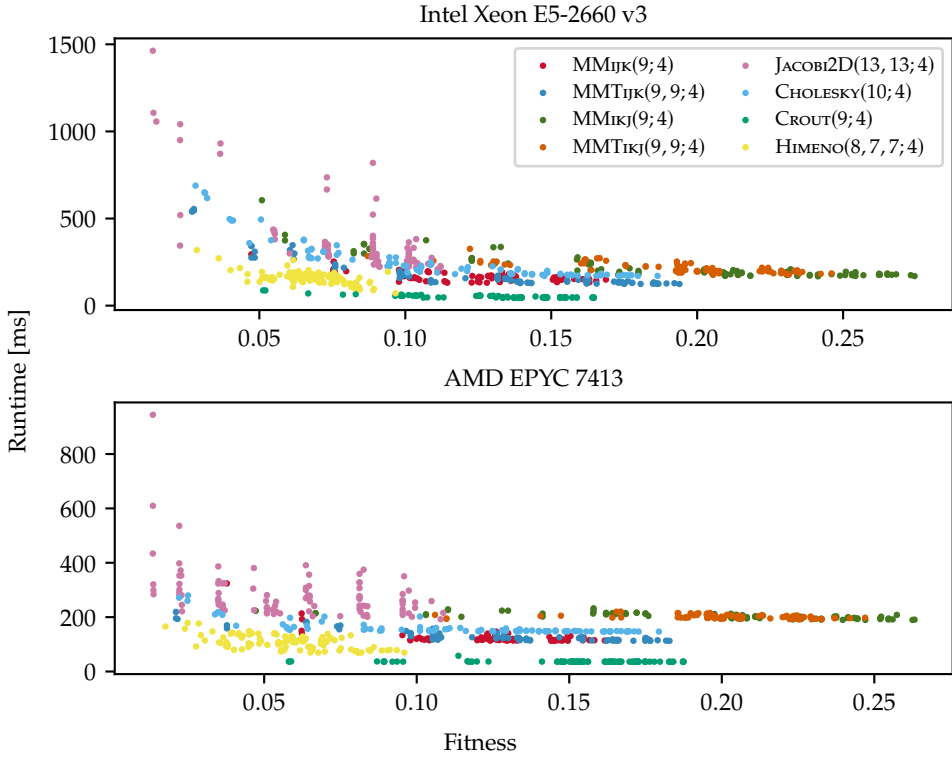


Figure 6.4: Scatter plot of the fitness and measured running time on an Intel Xeon E5-2660 v3 CPU and an AMD EPYC 7413 CPU for randomly chosen array layouts.

fitness than the canonical layouts. To this end, we perform the evolutionary process for each combination of our two simulated processors and eight access patterns, giving rise to a total of sixteen experiments. For all of these experiments, we configure our genetic algorithm to use $\mu = 20$, $\lambda = 20$, and a mutation rate of 25 %. We simulate a total of 20 generations in each case.

Figure 6.5 shows a violin plot of the fitness distribution of all individuals considered during the evolutionary process. Figure 6.6 shows the evolution of population fitness over the course of our experiments. Note that each of these experiments represents a single evolutionary process. We notice that for the MMTijk, MMikj, JACOBI2D, and HIMENO access patterns, our method does not manage to discover any layouts with higher fitness than the initial population of canonical layouts. In the experiment on the MMijk access pattern, we discover layouts with a fitness 149.8 % higher than the canonical layouts on the Intel Xeon E5-2660 v3 processor,

Table 6.2: Pearson’s coefficient of correlation (ρ_p) and Spearman’s coefficient of rank correlation (ρ_s) between our simulation-based fitness function and true running time.

Access pattern	Intel E5-2660 v3		AMD EPYC 7413	
	ρ_p	ρ_s	ρ_p	ρ_s
MMIJK(9; 4)	−0.672	−0.480	−0.648	−0.489
MMTIJK(9, 9; 4)	−0.810	−0.896	−0.863	−0.823
MMIKJ(9; 4)	−0.845	−0.815	−0.800	−0.838
MMTIKJ(9, 9; 4)	−0.777	−0.744	−0.291	−0.405
JACOBI2D(13, 13; 4)	−0.760	−0.769	−0.390	−0.428
CHOLESKY(10; 4)	−0.827	−0.953	−0.725	−0.892
CROUT(9; 4)	−0.846	−0.663	−0.213	−0.704
HIMENO(8, 7, 7; 4)	−0.607	−0.475	−0.561	−0.496

and we improve on the fitness of canonical layouts by 187.5 % for the AMD EPYC 7413. We also find layouts with improved fitness for the MMTIKJ (109.6 % and 141.1 % for the Intel and AMD processors, respectively), CHOLESKY (26.4 % and 36.8 %), and CROUT (545.9 % and 541.1 %) access patterns. It is notable that we are able to find layouts with high fitness in few generations.

6.5.4 Real-World Performance

In order to evaluate whether the layouts identified by our evolutionary algorithms as superior to canonical layouts are indeed better, we evaluate them on real hardware. We collect the fittest individual from each of the successful evolution experiments – i.e. experiments in which our method improved upon canonical layouts, as indicated by the top boundary in Figure 6.6 exceeding the maximum fitness in the first generation – and evaluate the performance of those layouts compared to the canonical layouts on real hardware. Given that our genetic algorithm discovered superior layouts for four access patterns – MMIJK, MMTIKJ, CHOLESKY, and CROUT – and that we evaluate a discovered layout and two canonical layouts for each access pattern, this gives rise to twenty-four experiments. We repeat each experiment ten times to compensate for run-to-run variance.

The results of our experiments are shown in Table 6.3; they show that some access patterns – the CHOLESKY pattern in particular – benefit very little from our method,

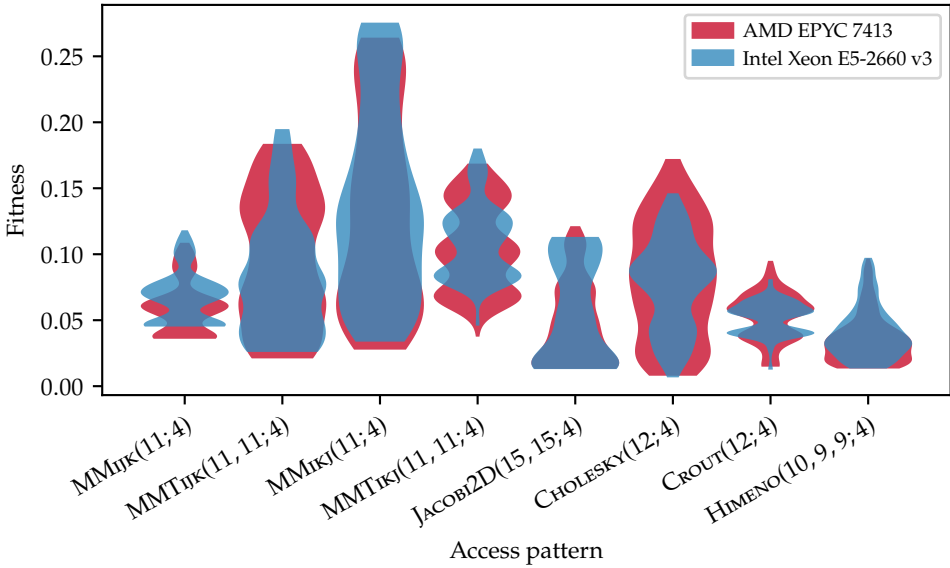


Figure 6.5: Distribution of the fitness values for all individuals found across evolution experiments for eight access patterns and two processors.

with speed-ups ranging from small on the Haswell processor to insignificant on the Zen 3 processor. The matrix multiplication access patterns benefit more, and performance for these access patterns is improved significantly. The CROUT access pattern stands out as achieving very large speed-up – up to a factor ten – from our method. It is worth noting that, in most cases, the Zen 3 processor benefits more from our evolutionary methodology than the Haswell processor; we do not currently have a satisfactory explanation for this behaviour.

It is important to note that we do not claim to have discovered a novel way of performing matrix multiplication or matrix decomposition that outperforms existing implementations. Indeed, our experiments are based on relatively naive implementations of these algorithms; high-performance implementations of matrix multiplication commonly rely on tiling to significantly improve the cache behaviour of the application [199], and the performance of tiled matrix multiplication surpasses what we achieve in this chapter. The purpose of the methodology described in this chapter, rather, is to provide an alternative way of improving the cache behaviour of an application in a manner which is fully agnostic of the application: unlike tiling and other application-specific optimisations, our methodology of altering the array layouts can be applied to any multi-dimensional problem without the need for application-specific knowledge. In addition, our approach

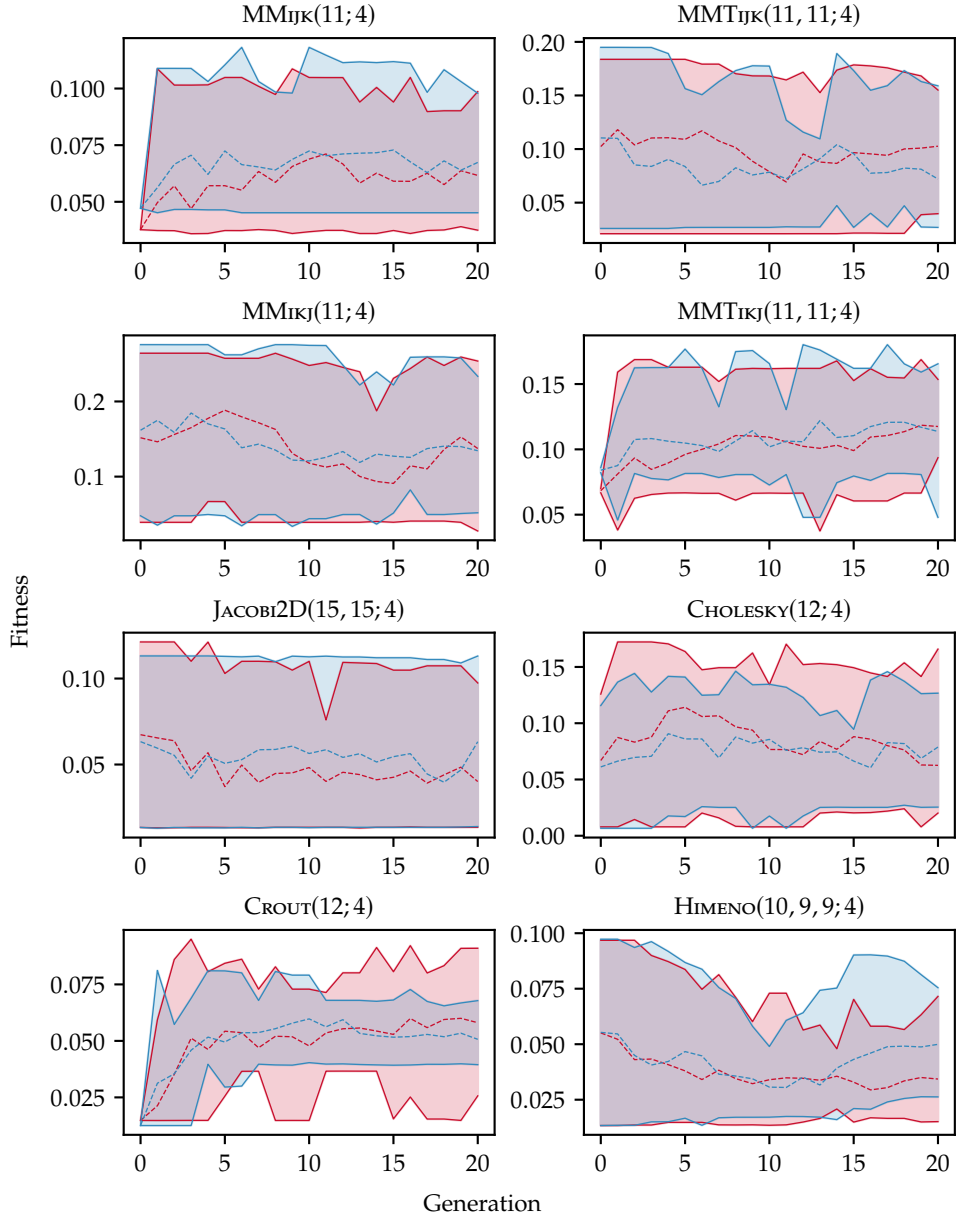


Figure 6.6: Range of fitness values across eight experiments for the Intel Xeon E5-2660 v3 (blue) and AMD EPYC 7413 (red). Mean fitness values are given by the dashed lines.

Table 6.3: Comparison of running time between the best-performing canonical layout and the best-performing layout found by our evolutionary process for four access patterns.

Access pattern	Best can.	Best evo.	Speed-up
Intel Xeon E5-2660 v3			
MMIJK(11; 4)	17.84 s	10.94 s	63.1 %
MMTIKJ(11, 11; 4)	18.13 s	13.96 s	29.9 %
CHOLESKY(12; 4)	11.84 s	11.43 s	3.6 %
CROUT(12; 4)	158.54 s	43.72 s	262.6 %
AMD EPYC 7413			
MMIJK(11; 4)	37.71 s	9.58 s	293.8 %
MMTIKJ(11, 11; 4)	32.35 s	15.21 s	112.6 %
CHOLESKY(12; 4)	9.72 s	9.55 s	1.0 %
CROUT(12; 4)	232.84 s	21.03 s	1007.0 %

requires few code changes, making it easy to implement.

6.6 Limitations and Threats to Validity

Throughout this chapter, we evaluate cache efficacy through a simplified lens which may reduce the applicability of our methods in more complex, real-world applications. Indeed, we consider accesses to memory in isolation, decoupled from computation and cache-polluting effects. We assume single-threaded execution without scheduling, which means that our caches will not be polluted by processes sharing (parts of) the cache hierarchy, nor will the application have its cached data evicted due to context switching. We also assume scalar, in-order execution of memory accesses. Finally, we take an optimistic view of cache latencies, using the fastest load-to-use latencies provided by hardware manufacturers; in real-world scenarios, cache latencies may be both more pessimistic and less stable than we assume. The results shown in Section 6.5.4 indicate, however, that our fitness function is sufficiently accurate to be effective in real hardware.

In addition, the family of array layouts described in this chapter requires array sizes to be powers of two in each dimension. In applications where this is not the case, arrays must be over-allocated. For n -dimensional applications, using the layouts described in this chapter requires over-allocation by a factor of $O(2^n)$.

Furthermore, applications using such layouts must consider the use of SIMD vectorisation: it remains an open question which operations on arrays laid out in non-standard ways can be (automatically) vectorised. We have argued for the feasibility of SIMD in Morton-like arrays in Section 6.3.4.

Finally, our work considers only multiset permutations, in which the rank significance of bits in the input indices is preserved. This decision is based on current commodity hardware, which is capable of efficiently permuting bits only under this condition. There exists an even larger family of layouts in which rank bit significance is not preserved⁴; such layouts could be of practical use in theoretical future processors with more advanced bit manipulation instructions, or in current FPGA and ASIC devices which permit the implementation of custom bit manipulation operations. Although we have not tested our approach on this further generalisation, we are confident that an evolutionary approach like the one presented in this chapter could be beneficial in exploring this (even larger) design space.

6.7 Reproducibility and Reusability

The evolutionary algorithms, scripts for the processing and visualisation of data, and other software used in this chapter are permanently archived on Zenodo [256], and have been made available at doi:10.5281/zenodo.10567243. The aforementioned artifact also contains all data that were gathered and processed during the work presented in this chapter as well as a README file explaining its use. The artifact was submitted to the artifact evaluation track of the ICPE'24 conference, where it was awarded the *ACM Artifacts Available*, *ACM Artifacts Evaluated – Functional*, and *ACM Results Validated – Reproduced* badges. At the time of writing, the software – named ALEX for *Array Layout EXperiment* – is available freely under the Unlicense on GitHub at <https://github.com/stephenswat/alex> [257].

6.8 Summary

In this chapter, we have discussed a generalisation of the Morton layout for multi-dimensional data and we have shown that there exist families of array layouts with strongly varying cache behaviour which, in turn, impact the performance of applications. We have shown how these layouts can be systematically described,

⁴That is to say, the layout $[0_0, 0_1, 1_0, 1_1]$ (which draws its least significant bit from the least significant bit of the first index) is distinct from the layout $[0_1, 0_0, 1_0, 1_1]$ (which instead draws its least significant bit from the *second-least* significant bit of the first index).

and that the number of possible layouts quickly exceed the limits of what can be feasibly explored using exhaustive search. We have proposed a method based on evolutionary algorithms for the exploration of the design space of such layouts. We have evaluated the fitness of different array layouts using cache simulation and we have presented results indicating that our fitness function correlates with real world performance. Furthermore, we have shown that the methodology described in this chapter can be used to improve the performance of applications on real hardware by up to ten times compared to the canonical row-major and column-major layouts. Finally, we have further expanded on our answer to Research Question 2.

Modelling and Mitigating Thread Imbalance in SIMT Workloads

Remember that all models are wrong; the practical question is how wrong do they have to be to not be useful.

— **George Box**
(Professor of statistics)

The second common performance problem we identified in Chapter 4 is imbalance in computational workloads. Although multi-core CPU architectures can execute workloads in which threads perform different amount of work with relatively few problems, the lockstep execution behaviour of many massively parallel architectures dictates that each thread in a thread group must wait for the thread with the most work to finish. This can introduce significant overhead, but it is not trivial to estimate exactly how much. In this chapter, we answer Research Question 3 by introducing a statistical model that gives a metric for the amount of overhead incurred by lockstep execution for a workload following a given statistical distribution. In other words, we provide a metric for the degree to which a workload is suitable for massively parallel execution. We show that our metric can be used to evaluate thread coarsening and refinement techniques as well as load balancing in early stages of the development of parallel applications. Furthermore, we show that our model closely matches the behaviour of real hardware.

This chapter is based on the following publication:

- Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Attila Krasznahorkay and Andy D. Pimentel. ‘Modelling Performance Loss due to Thread Imbalance in Stochastic Variable-Length SIMT Workloads’. In: *2022 30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. MASCOTS’22. Nice, France: IEEE, 2022, pp. 137–144. DOI: 10.1109/MASCOTS56607.2022.00026

7.1 Introduction

As the landscape of high-performance computing has evolved over recent years, Single Instruction Multiple Threads (SIMT) processors – usually in the form of General-Purpose Graphics Processing Units (GPGPUs) – have become popular for high-performance computation in many domains [258]. By sacrificing the independence of individual processing cores, SIMT processors are able to pack significantly more processing cores, and thus provide much more raw processing power, compared to their traditional Multiple Instructions Multiple Data (MIMD) counterparts [107].

However, not every conceivable computational workload can be efficiently handed off to an SIMT device. The increased raw processing power of these devices comes at the cost of reduced flexibility, and algorithms must be carefully designed to run efficiently on SIMT devices, lest their computational prowess goes to waste. One important consideration when programming SIMT devices is the concept of *thread divergence*. In an SIMT device, a group of threads can – by definition – perform only a single, common instruction at a time; colloquially, these threads run in *lockstep*. Thus, cases where the execution paths of threads diverge will cause some of the threads to be idle. If care is not taken to minimise thread divergence in algorithms designed to run on SIMT devices, it can severely degrade performance [259].

Thread divergence emerges not only in situations with conditional branches in the common *if-else* sense, but it can also arise in iterative processes in the form of *thread imbalance*. When the number of iterations of a loop varies between threads, the result is divergence: threads will be idle until the thread with the largest amount of work has performed the necessary number of iterations. Throughout this chapter, we refer to workloads where the number of iterations is not fixed and may differ between threads as *variable-length workloads*. When the number of iterations is described by some probabilistic process, we refer to them as *stochastic workloads*. While it is well understood that thread imbalance in variable-length workloads is detrimental to the performance of SIMT devices [259, 260], we are unaware of any quantitative models that predict exactly how much performance is lost.

The question how we can model the impact of thread imbalance in stochastic variable-length workloads is the core focus of this chapter. With this chapter, we are the first to design and implement an accurate statistical model for the expected performance loss of a given application, given *only* that it is an iterative process,

that it is executed on an SIMT device, and that the number of iterations required to complete the process follows a known (albeit arbitrarily complex) distribution. We validate our model using empirical measurements gathered using a dedicated benchmark running on an NVIDIA GPU. The results of this validation show that our model agrees with simulated data with a relative error of less than 0.1 %, and that it agrees with measurements taken on a real device within 2 %.

Our accurate model can help motivate more precisely the design process of (future) SIMT applications – in particular in terms of processing granularity – in domains where stochastic iterative processes are common, such as machine learning [261], cryptography [262], graph processing [263], and scientific computing [264]. The importance of thread imbalance and granularity is further supported by our own results, which show (in Table 7.1) that thread imbalance in SIMT devices can lead to execution that is nearly four times slower if thread granularity is not chosen carefully.

In short, this chapter makes the following contributions:

- We provide a statistical framework for reasoning about the performance loss due to thread imbalance in variable-length workloads on SIMT processors (Section 7.4);
- We assess the accuracy of our model using empirical results gained from a custom synthetic benchmark for this form of performance loss (Section 7.5);
- We demonstrate through a mini-app how our model can be used to inform the design of real-world applications (Section 7.6).

7.2 Background

A traditional multi-core CPU architecture consists of a number of processing cores, each of which is equipped with dedicated arithmetic and control hardware. Because all cores in such an architecture possess their own control, they can function largely independently of one another, executing multiple instructions on (potentially) different data. Thus, such an architecture is classified as Multiple Instructions Multiple Data (MIMD) [77]. In recent years, we observe a stark rise in popularity of a different kind of architecture: Single Instruction Multiple Threads (SIMT). SIMT architectures omit the control flow from individual cores in favour of having a larger number of (less flexible) cores and – as a result – more arithmetic prowess compared to a similar MIMD device.

	m_0	m_1	m_2	m_3	t_0	t_1	t_2	t_3
<code>int n = thread_id();</code>	✓	✓	✓	✓	⤵	⤵	⤵	⤵
<code>prologue();</code>	✓	✓	✓	✓	⤵	⤵	⤵	⤵
<code>if (0 < n < 3) {</code>								
<code>branch1();</code>	✗	✓	✓	✗	⤵	⤵	⤵	⤵
<code>} else if (n == 0) {</code>								
<code>branch2();</code>	✓	✗	✗	✗	⤵	⤵	⤵	⤵
<code>}</code>								
<code>epilogue();</code>	✓	✓	✓	✓	⤵	⤵	⤵	⤵

Figure 7.1: A toy program that is executed by four threads – with identifiers 0, 1, 2, and 3 – which shows masking and thread divergence. Thread t_i executes instructions only if mask m_i is set. Idle threads are represented by grey arrows, while active threads are represented by black arrows.

7.2.1 Thread Divergence and Imbalance

In an SIMT architecture, multiple threads – which we refer to as a *thread group* – share the same control flow. As a result, instructions on such architectures can only ever be issued to an entire thread group at the same time, rather than to individual threads like on an MIMD architecture. This behaviour is referred to as *lockstep* execution. In this execution model, conditional branches are challenging and are implemented through a process known as *masking*: when a thread group encounters a conditional block, each thread determines whether it should execute or ignore the corresponding instructions. An example of masked execution is shown in Figure 7.1. Threads not participating in conditional execution are unable to perform other useful work during this time: they are idle, and computing resources are wasted. As the number of conditional paths – or the length of those paths – grows, more threads are idle for longer periods of time, and we lose more performance. We refer to this behaviour as *thread divergence*, and it can be a significant source of performance degradation [259].

Thread divergence also arises in iterative structures such as loops, which rely on conditional branches that jump back to the start of the loop body. Thus, if one thread in an SIMT processor has concluded the iterative process but another thread has more iterations to perform, their execution paths diverge: the first thread will need to idle until the second completes the loop. We refer to this particular manifestation of threads divergence as *thread imbalance*.

7.2.2 Notation

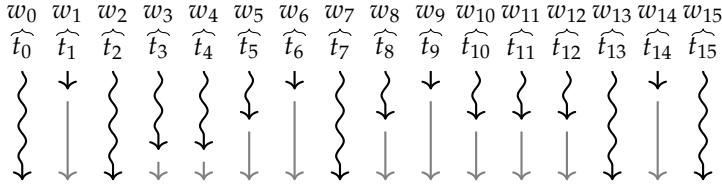
The remainder of this chapter relies heavily on concepts from the field of statistics, for which we will adhere to a common system of notation: capital letters (W, X, Y, Z) shall denote (possibly parameterised) random variables, $f_X(x)$ shall denote the probability mass function of X , and $F_X(x)$ shall denote the cumulative density function of X . $\text{supp}(X)$ shall denote the support of X . $X_{(i:n)}$ shall denote the i -th order statistic of a sample of n values drawn from X . $P(\dots)$ shall denote the probability of a given event.

We will use terminology from NVIDIA's CUDA framework for programming GPUs [97], as this is one of the most widely used SIMT programming platforms at the time of writing. In addition, the experiments presented in this chapter are designed for and executed on NVIDIA GPUs. In particular, we will use the term *warp* to refer to a group of threads running in lockstep.

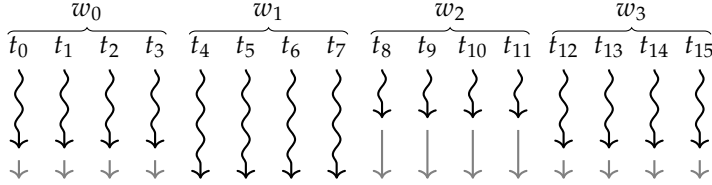
7.3 Mitigating Imbalance

The SIMT programmer's toolbox provides at least two strategies to ameliorate the effects of thread imbalance: changing the thread granularity, and load balancing. Thread granularity refers to the way work is mapped onto the threads of the processor. Most commonly, SIMT programmers map small, independent parts of a workload onto individual threads, but it is also possible to spread that work over multiple threads. This effectively reduces the number of independent jobs executed by the thread group, thereby reducing the degree of imbalance. This process is referred to as *thread refinement*, as it maps a smaller amount of work onto each thread; Figure 7.2 illustrates three levels of thread refinement. Spreading work over multiple threads is often non-trivial and implementing thread refinement can be challenging. An opposite approach known as *thread coarsening* can also be used; coarsening involves *increasing* the workload per thread in the hope that this will balance the amount of work between threads. We note that support for programming at different levels of granularity is increasing in massively parallel programming models; NVIDIA, for example, implements so-called *cooperative groups* in version 9.0 of its CUDA platform [97, 265]. Similarly, SYCL features *sub-groups* [266]. Both of these features allow programmers to tune their code to minimise the impact of thread imbalance.

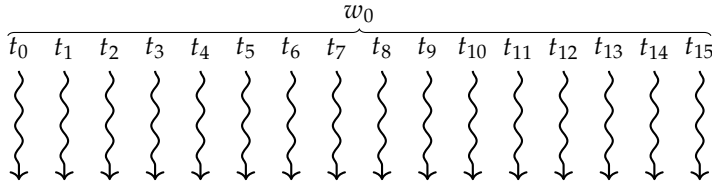
Load balancing, on the other hand, operates by pre-processing the workload of the SIMT processor such that the work performed by each thread group is more



- (a) Mapping one unit of work onto one thread risks thread imbalance, but allows the code to be written in a sequential fashion.



- (b) Mapping one unit of work onto multiple threads reduces thread, but requires the programmer to distribute work over multiple threads.



- (c) Mapping one unit of work onto an entire group (warp) of threads eliminates thread imbalance, but requires the greatest effort to divide work.

Figure 7.2: Three distinct levels of thread refinement which can be used to mitigate the effects of thread imbalance.

balanced [263]. For example, a programmer may choose to sort the workload before offloading it to the SIMT device, guaranteeing that jobs of similar length will end up in the same thread group. Of course, sorting a sufficiently large workload is in itself a costly operation, and approximate solutions – which nevertheless more balanced execution – may be used [267].

Choosing a suitable thread granularity or balancing loads between SIMT threads requires an understanding of how much performance we lose due to imbalance, but we do not currently have quantitative models for this. Rather, these important (and potentially time-consuming) optimisations are often processes of trial and error. Our contribution, therefore, is to provide a quantitative model for the impact of thread imbalance in order to allow application developers to make more informed decisions about the design of their applications.

7.4 Modelling Imbalance

Throughout this chapter we assume that a workload is comprised of an arbitrary number of *units of work* which can be performed independently and in parallel. Each unit is described by a natural number, indicating the number of iterations required to complete it. For example, the workload $\{5, 2, 3, 7\}$ consists of four units of work, requiring 5, 2, 3, and 7 iterations to complete. Executing one iteration has some application-specific computational cost T ; this might represent, for example, a number of cycles or an amount of wall-clock time. It follows that the computational cost of a unit of work is given by the number of iterations required to complete it multiplied by the iteration cost T .

As workloads can be arbitrarily large and hardware is inherently limited in its ability to execute processes in parallel, the workload is naturally partitioned into *work groups*. On an SIMT device with two lanes, our previous workload might be partitioned into two work groups: $\{\{5, 2\}, \{3, 7\}\}$. Work groups are analogous to thread groups in the same way that data is analogous to hardware; work groups are mapped onto thread groups for processing, and a single thread group may process many work groups throughout the running time of a program. The mapping of individual units of work onto individual threads is determined by the thread granularity. The manner in which the work groups within a workload are mapped onto thread groups depends on the hardware, and has no bearing on the rest of our model: on a strict SIMT device, work groups might be executed sequentially by a single thread group, while on an NVIDIA GPU the work groups might be executed in parallel across multiple independent Streaming Multiprocessors (SMs).

Importantly, the idea of imbalance exists only *within* work groups, and there is (by definition) no dependency between different work groups. This allows us, without loss of generality, to study work groups individually. As such, we denote work groups using the symbol w , and we use $|w|$ to denote the size of work group w . Finally, w_i shall denote the number of iterations required to complete the i th unit of work in w .

We define performance loss due to SIMT execution as the computational cost of executing a work group w on a SIMT device, $C_{\text{SIMT}}(w)$, divided by the cost of executing the same work group on an idealised MIMD device, $C_{\text{MIMD}}(w)$. This idealised device has equivalent computational power to the SIMT device, but does not run in lockstep and as such, its performance does not degrade due to thread imbalance. Figure 7.3 illustrates the execution of a workload on these two devices. Formally, the performance loss of executing a work group w on the SIMT device,

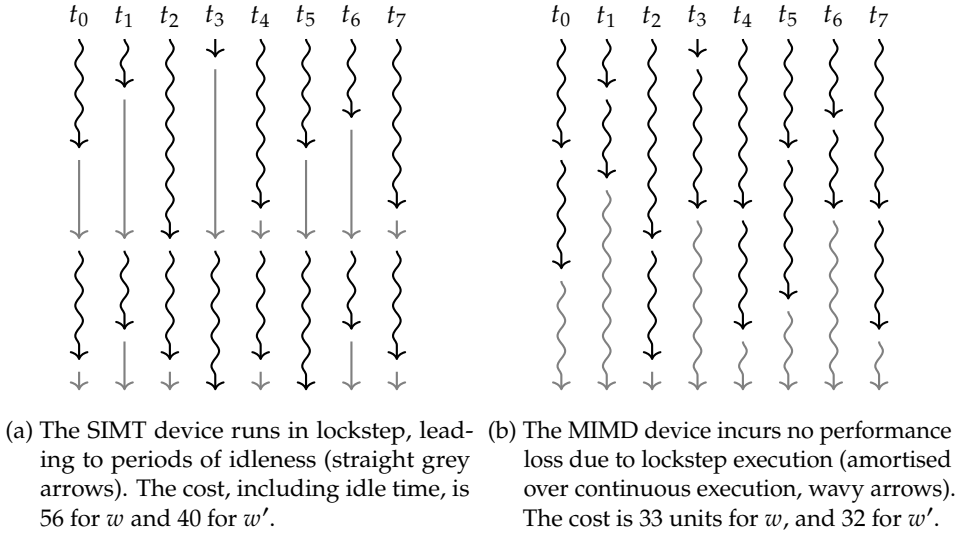


Figure 7.3: An example of 8-way SIMT and MIMD execution of two work groups from a larger workload: $w = \{4, 2, 7, 1, 6, 4, 3, 6\}$ and $w' = \{4, 3, 4, 5, 4, 5, 3, 4\}$. The performance loss due to imbalance is $\mathcal{H}(w) = 56/33 = 1.70$ and $\mathcal{H}(w') = 40/32 = 1.25$.

$\mathcal{H}(w)$, is defined as:

$$\mathcal{H}(w) = \frac{C_{\text{SIMT}}(w)}{C_{\text{MIMD}}(w)} \quad (7.1)$$

From our assumptions made about the devices, it follows that $\mathcal{H}(w) \in [1, \infty)$; indeed, since the MIMD device has the same computational power, but is not affected by thread imbalance, it should always perform as well as or better than the SIMT device. In this framework, $\mathcal{H}(w) = 1$ implies that the computational cost of running the work group on the SIMT device is the same as the cost of running it on the MIMD device, and indicates that the work group incurs no performance loss at all. Intuitively, as $|w|$ becomes larger (in other words, as we process more work in parallel), we expect the performance loss to grow accordingly. Finally, $|w| = 1$ implies $\mathcal{H}(w) = 1$, as there is no possibility for a single unit of work to be imbalanced.

7.4.1 Modelling SIMT and MIMD Devices

In order to model the computational cost of executing a work group on our SIMT device we must consider the fact that, in lockstep execution, a thread cannot proceed until all threads in its group have completed their required number of iterations. Thus, the threads need to wait for the thread with the largest number of iterations: the *depth* of the work group [268]. Because *all* threads in the group are occupied (albeit possibly idle) throughout the entire process, the total computational cost for the work group w , $C_{\text{SIMT}}(w)$, is given by the following expression, where T represents the computational cost of executing a single iteration:

$$C_{\text{SIMT}}(w) = |w| \max_{i=1}^{|w|} T w_i = T |w| \max_{i=1}^{|w|} w_i \quad (7.2)$$

Next, we model the computational cost of executing the same work group on our idealised MIMD device. This device executes the units of work in parallel, but can immediately perform meaningful work when previous work is completed such that threads do not incur additional cost by idling. Therefore, the computational cost of executing the work group on our MIMD device is equal to the *sum* of the costs of the individual units. This gives us the following definition of $C_{\text{MIMD}}(w)$, where the cost T to perform one iteration is the same as for the SIMT device:

$$C_{\text{MIMD}}(w) = \sum_{i=1}^{|w|} T w_i = T \sum_{i=1}^{|w|} w_i \quad (7.3)$$

We can substitute Equations 7.2 and 7.3 into Equation 7.1 to obtain the following expression for the loss of performance:

$$\mathcal{H}(w) = \frac{T |w| \max_{i=1}^{|w|} w_i}{T \sum_{i=1}^{|w|} w_i} = |w| \frac{\max_{i=1}^{|w|} w_i}{\sum_{i=1}^{|w|} w_i} \quad (7.4)$$

It is worth noting that, in Equation 7.4, the constant cost factor T is eliminated, which imparts a powerful property on our model: it is wholly independent of the implementation details of the iterative code, as well as the hardware on which it will run. This means that no knowledge about the implementation is required to construct a model of this type. Rather, we only need to know how the number of iterations for each work unit is distributed.

7.4.2 Modelling Stochastic Work Groups

From this point forward, we treat our work groups as random samples, which necessitates some change in notation. In a stochastic framework, our work group w becomes a realisation of an independently and identically distributed sample of size $n = |w|$ drawn from a discrete distribution W , such that $w = \{W_1, W_2, \dots, W_n\}$ and $W_1, W_2, \dots, W_n \sim W$. Thereby, $\mathcal{H}(w)$ necessarily becomes a random variable, which we shall denote $X_W(n)$, such that $X_W(n) \sim \mathcal{H}(\{W_1, W_2, \dots, W_n\})$. This gives the following rephrasing of Equation 7.4 in terms of random variables:

$$X_W(n) = n \frac{\max_{i=1}^n W_i}{\sum_{i=1}^n W_i} \quad (7.5)$$

The idea that iteration counts are drawn from or described by a statistical distribution arises naturally in many kinds of computation. Programs which model or process data from real-world processes might be described – with sufficient domain knowledge – by one of many well-known parametric distributions, such as the Poisson distribution. In many other cases where a closed-form distribution is not available, we can still describe the iteration-counts distribution through simulation.

This stochastic framework also elucidates some of the potential use cases for our model. Indeed, the idea of configuring the thread granularity of a program corresponds directly to changing the size n of the random sample of work lengths, and load balancing corresponds to changing the underlying distribution W ; balancing the load of a SIMT program amounts to partitioning it into multiple smaller loads, each of which has a more narrow distribution and – as a result – loses less performance due to imbalance. Because our model captures these effects so directly, we believe it to be a useful tool in reasoning about these kinds of optimisations.

Given that the distribution of our work unit lengths, W , is discrete, we could – in theory – find the distribution of the performance loss $X_W(n)$ by enumerating all possible values of W_1, W_2, \dots, W_n . However, this solution runs in $\mathcal{O}(|\text{supp}(W)|^n)$ time and is not generally computationally tractable: a distribution supported by only ten possible outcomes would lead to 10^{32} possible combinations in a model for thirty-two parallel units of work.

We proceed by creating an equivalent model that is more computationally efficient. In order to do so, we first re-write the numerator in Equation 7.5 using order statistics: given that W_1, W_2, \dots, W_n are random variables drawn from W , we can

sort these values in ascending order such that the i th order statistic, denoted $W_{(i:n)}$, is the k th smallest value. Thus, $W_{(1:n)}$ represents the smallest value in the sample, $W_{(2:n)}$ represents the second-smallest value, and so forth. Given that there are n values in total, $W_{(n:n)}$ naturally represents the largest value in the sample – the maximum. Next, we reduce the denominator of the fraction to a single random variable. This new random variable, denoted $Z_W(n, a)$, represents the distribution of the sum of n random variables drawn from W , given the fact that the maximum of that sample is known to be a . Since the maximum value is known from the numerator, we set $a = W_{(n:n)}$ and obtain the following equation:

$$X_W(n) = n \frac{a}{Z_W(n, a)} \text{ where } a \sim W_{(n:n)} \quad (7.6)$$

The number of random variables required to compute the performance loss $X_W(n)$ in Equation 7.6 is now only two (as opposed to the n random variables required to compute Equation 7.5): $W_{(n:n)}$ and $Z_W(n, W_{(n:n)})$. This greatly reduces the combinatorics required to enumerate all possible outcomes of this division; indeed, we find that the number of outcomes is now bound by $\mathcal{O}(n|\text{supp}(W)|^2)$. We proceed by computing the distribution of the sample maximum $W_{(n:n)}$ in Section 7.4.2, and the distribution of the sum of a sample given its maximum, $Z_W(n, a)$, in Section 7.4.2. Finally, we calculate the ratio distribution $X_W(n)$ in Section 7.4.3.

Distribution of the Sample Maximum

In order to find an analytical solution for the distribution of the maximum of a series of i.i.d. random variables $W_1, W_2, \dots, W_n \sim W$, we note that this maximum is equal to the n -th order statistic of that sample, denoted $W_{(n:n)}$. The distribution of order statistics for discrete distributions is well understood [269], and the distribution of $W_{(n:n)}$ is described by the following probability mass function:

$$\begin{aligned} f_{W_{(n:n)}}(x) &= P(W \leq x)^n - P(W < x)^n \\ &= F_W(x)^n - (F_W(x) - f_W(x))^n \end{aligned} \quad (7.7)$$

It is worth noting that the distribution of the maximum value preserves the support of the original distribution; intuitively, if one were to roll a six-sided die ten times and select the maximum roll, that outcome would never be more than

six, nor would it ever be lower than one.

Distribution of the Sample Sum

In Equation 7.6, $Z_W(n, a)$ denotes the distribution of the sum of an i.i.d. sample $W_1, W_2, \dots, W_n \sim W$ given a priori knowledge that the maximum value in that sample is equal to a , thus:

$$f_{Z_W(n,a)}(x) = P\left(x = \sum_{i=1}^n W_i \mid W_{(n:n)} = a\right) \quad (7.8)$$

In order to derive this distribution we construct a parametric, *non-normalised* function $g_W(x; i, m)$, which denotes the probability of the sum of i values drawn from W , each of which is no greater than m , being equal to x . This function is defined inductively from a degenerate distribution supported at zero (capturing the idea that additive processes start at zero) as follows:

$$g_W(x; i, m) = \begin{cases} [x = 0] & \text{if } i = 0 \\ \sum_{j=0}^m g_W(x - j; i - 1, m) f_W(j) & \text{otherwise} \end{cases} \quad (7.9)$$

By its definition, $g_W(x; n, a)$ nearly models the target distribution but it crucially fails to model that, in for the maximum of a sample to equal a , the sample must contain a at least once. In order to resolve this, we subtract the function $g_W(x; n, a - 1)$, which intuitively models the probability of *never* drawing a , point-wise. Then, we only need to normalise the resulting probabilities to find an expression for the distribution of the sample sum given its maximum:

$$f_{Z_W(n,a)}(x) = \frac{g_W(x; n, a) - g_W(x; n, a - 1)}{\sum_{j=a}^{na} (g_W(j; n, a) - g_W(j; n, a - 1))} \quad (7.10)$$

7.4.3 Modelling Performance Loss

In order to find the distribution of the loss of performance in our model, we calculate the ratio distribution between $W_{(n:n)}$ and $Z_W(n, W_{(n:n)})$. This process does not generally permit a closed form solution, but this distribution can be calculated through a brute-force approach. Indeed, since all distributions involved in the process are discrete and because their supports are subsets of the integers, our ratio distribution is supported by a subset of \mathbb{Q} which must be countable. The probability of each supported outcome is then given by summing up the

probabilities of all outcomes which map to the same irreducible fraction:

$$f_{X_W(n)}(x) = \sum_{a,b \in \mathbb{N}, n \frac{a}{b} = x} f_{W(n:n)}(a) f_{Z_{W(n,a)}}(b) \quad (7.11)$$

It should be noted that, for this process to be tractable, the support of the underlying distribution W must be finite. If the support of W is infinite, then the support of $W_{(n:n)}$ is infinite and, consequently, the support of $X_W(n)$ is also infinite; as such, we would not be able to meaningfully enumerate the possible outcomes. We can resolve this issue by approximating W using a finite truncated distribution. To this end, we can choose some acceptable degree of error $\epsilon = 0.001$ and construct a new distribution W' where:

$$\text{supp}(W') = \{x \in \text{supp}(W) : F_W(x) \leq 1 - \epsilon\} \quad (7.12)$$

Following normalisation, distribution W' would capture roughly $1 - \epsilon$ of the total probability mass of the original distribution, while allowing us to enumerate the possible outcomes in a finite amount of time.

The enumeration of the possible outcomes, and the computation of their probabilities, marks the end of the construction of our model. This model gives us insight into the distribution of the loss of performance for a randomly drawn work group. For example, $f_{X_W(n)}(2/1)$ gives us the probability that an arbitrary work group would have a computational cost exactly twice as high when executed on an SIMT device compared to when executed on an MIMD device. Similarly, $F_{X_W(n)}(2/1)$ denotes the probability of that performance lost being equal to or less than a factor two. Additionally, $\mathbb{E}(X_W(n))$ gives the expected loss of performance for a work group. Because most real-world workloads will have many thousands of work groups, the performance loss of such workloads will naturally tend towards the expected value as a consequence of the law of large numbers.

7.4.4 Approximate Models

The modelling method presented in this chapter is designed explicitly to produce a model that captures the distribution of overhead exactly and within a tractable amount of time. However, we acknowledge that in certain scenarios, the time required to construct such a model may still be prohibitive. In particular, we consider cases where the number of parallel processes is very large (much larger than the thirty-two threads which we see in NVIDIA devices [97]).

In cases where the number of parallel processors is very large, much of the

complexity in our model exists to capture the non-independence between the numerator and the denominator in Equation 7.5. At the same time, however, we find that models for higher numbers of threads are far more resistant to error that is introduced by a more lax treatment of the aforementioned non-independence. In other words, we find that approximate models that do not capture the non-independence in Equation 7.5 exactly may still give viable results for high degrees of parallelism. It is possible to create an approximate model by rewriting Equation 7.5 as follows:

$$X = n \frac{Y_{(n)}}{Y_{(n)} + \sum_{i=1}^{n-1} Y_{(i)}} \quad (7.13)$$

By treating the summation in the denominator of this equation as independent from the numerator, we can independently draw a new sample $Z_1, Z_2, \dots, Z_{n-1} \sim Z$, where Z is a truncated version of the distribution Y , supported up to and including $Y_{(n)}$:

$$X = n \frac{Y_{(n)}}{Y_{(n)} + \sum_{i=1}^{n-1} Z_i} \quad (7.14)$$

The distribution for the sum in the denominator of this equation can be found by calculating the repeated self-convolution of Z , which in turn can be calculated as a power of the moment generating polynomial of Z [270]:

$$M_{\left(\sum_{i=1}^{n-1} Z_{(i)}\right)}(t) = M_Z(t)^{n-1} \quad (7.15)$$

This approximation provides excellent agreement for even a moderate number of threads (in our experience, the results are nearly indistinguishable if $n > 8$). For an even weaker but computationally cheaper approximation, we can treat the cost models for our SIMT and MIMD devices as completely independent, such that Equation 7.5 becomes equivalent to:

$$X = n \frac{Y_{(n)}}{\sum_{i=1}^n Y'_i} \quad (7.16)$$

Where $Y'_1, Y'_2, Y'_n \sim Y$. This allows us to estimate the ratio distribution from the distribution of the numerator given in Equation 7.7, while we can find the distribution for the denominator using convolution as described in Equation 7.15.

This approximation technique is computationally cheap, but produces highly skewed results where $Y < 1$ is possible (even though this should be impossible). Still, the mean is largely unaffected if the number of threads is sufficiently large.

7.5 Model Evaluation

In order to evaluate the predictive power of our model, we construct a benchmark based on a simple iterative process: matrix exponentiation. Our validation process has two stages. First, we simulate a synthetic workload – with exponents drawn from a given distribution – using a Monte Carlo method. This process allows us to compute the loss of performance as given by Equation 7.4 exactly, and we will refer to this as the *simulated* loss of performance. Second, we execute the matrix exponentiation kernel – with the iteration counts given by our earlier simulation – on a real-world SIMT device; by measuring the execution time of each of the work groups, we can determine the *measured* loss of performance. Both of these metrics can then be compared to each other, as well as to the *modelled* loss of performance, to confirm whether they are in agreement.

7.5.1 Benchmark Design

Our benchmark operates by computing powers of square dense matrices M^p (with $p \in \mathbb{N}$) through repeated multiplication¹. This problem matches the class of algorithms targeted by our model very well: each work unit iteratively executes matrix multiplication operations, and the number of iterations is equal to the exponent p . In addition, the cost of each iteration is fixed: for our benchmark, we operate on 16×16 matrices. As discussed in Section 7.4, the execution time of each individual step is irrelevant to the outcome of our model, and as such the size of the matrices should not matter. However, we find that if the run-time of each individual step is very small (as it is for, say, 3×3 matrices), we incur additional noise in our measurements. Pseudo-code for our benchmark is given in Algorithm 7.1.

Note that our validation strategy relies on an emulation of MIMD behaviour on the SIMT device. To this end, we measure the cost of performing a computation in accordance with the SIMT model as the time between the start of the computation and the time at which *all* threads are done. In contrast, we emulate the behaviour of

¹The astute reader may have noticed that, because square matrices under multiplication form a monoid, this operation can be performed more efficiently in $O(\log_2 p)$ time. However, this defeats the purpose of our benchmark, and is therefore not implemented.

Data: Source distribution W , number of trials r , work group size n

Result: Simulated results x_0, \dots, x_r and measured results y_0, \dots, y_r

```

for  $i \leftarrow 0$  to  $r$  do
  for  $j \leftarrow 0$  to  $n$  do
     $p_{i,j} \leftarrow W$ 
     $M_{i,j} \leftarrow$  random matrix
  end
end
for  $i \leftarrow 0$  to  $r$  do
   $C_{\text{SIMT}} \leftarrow 0$ 
   $C_{\text{MIMD}} \leftarrow 0$ 
  parallel for  $j \leftarrow 0$  to  $n$  do
     $M'_{i,j} \leftarrow I$ 
     $c_j \leftarrow \text{clock}()$ 
    for  $k \leftarrow 0$  to  $p_{i,j}$  do
       $M'_{i,j} \leftarrow M'_{i,j} M_{i,j}$ 
       $c_{\text{MIMD},j} \leftarrow \text{clock}()$ 
    end
     $c_{\text{SIMT},j} \leftarrow \text{clock}()$ 
     $C_{\text{SIMT}} \leftarrow C_{\text{SIMT}} + c_{\text{SIMT},j} - c_j$ 
     $C_{\text{MIMD}} \leftarrow C_{\text{MIMD}} + c_{\text{MIMD},j} - c_j$ 
  end
   $x_i \leftarrow n \frac{\max_{j=0}^n p_{i,j}}{\sum_{j=0}^n p_{i,j}}$ 
   $y_i \leftarrow \frac{C_{\text{SIMT}}}{C_{\text{MIMD}}}$ 
end

```

Algorithm 7.1: Benchmark of the loss of performance when running work units drawn from a given distribution. We assume execution on an SIMT machine – thus, the parallel block is executed in lockstep – to determine C_{SIMT} , and *emulate* the equivalent MIMD execution to calculate C_{MIMD} .

an MIMD device by calculating the time at which each thread is ready to proceed with further useful work, without being constrained by the lockstep execution model.

We evaluate the accuracy of our model using the following five underlying probability distributions for work unit lengths:

- $B(40, 0.5)$: Binomial with $n = 40$ and $p = 0.5$.
- $\text{Geo}(0.05)$: Geometric with $p = 0.05$.
- $\text{Pois}(30)$: Poisson with $\lambda = 30$.
- $U(20, 40)$: Uniform with $a = 20$ and $b = 40$.
- $\text{NB}(5, 0.3)$: Negative binomial with $r = 5$ and $p = 0.3$.

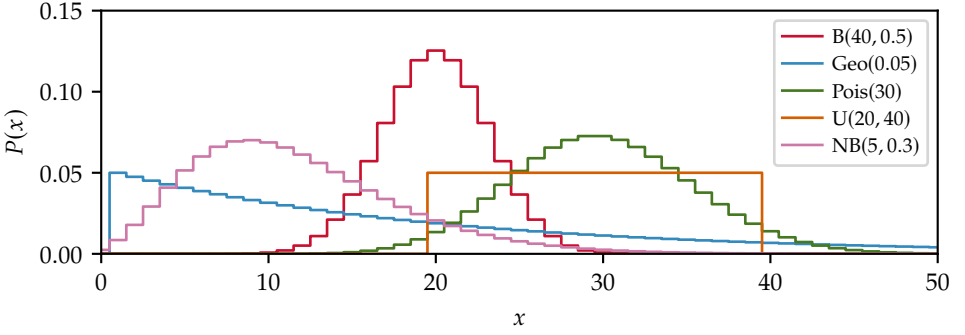


Figure 7.4: Probability mass functions for the distributions under evaluation in this chapter.

The probability mass functions for these distributions is given in Section 7.5.1. These distributions have been selected for evaluation because they: (1) occur naturally and commonly in real-world processes; (2) they have a wide range of supports (including infinite ones); and (3) they have a wide variety of shapes (including fat-tailed and thin-tailed). Please note that our model is not limited to such well-behaved distributions; instead, the model works for arbitrary discrete distributions. Even a categorical distribution assigning an arbitrary probability mass to each of a set of natural numbers can be used with our modelling strategy.

7.5.2 Experimental Setup

We have implemented our benchmark in C++; the Monte Carlo simulation of work items follows the MT19937 pseudo-random number generator provided by the C++ standard library [271]. The code for the SIMT device was written in CUDA [97], and it was compiled and executed on the CUDA 11.5 platform. The compiler was configured to emit code for Compute Capability and PTX version 8.0 (the most recent version supported by our target GPU). The host code was compiled using gcc version 9.4.0. Our results were generated on a node of the DAS-6 cluster [194] using an NVIDIA A100 PCIe GPU with 40 GB of HBM2 memory based on the *Ampere* microarchitecture [64]. The kernels were launched with 256 threads per block, and threads whose index exceeded the target number of threads were made to terminate immediately. This was done to reduce the occupancy of the Streaming Multiprocessors in an attempt to avoid unrealistic amounts of context switching. For each experiment, we used 2^{18} work groups to ensure the

Table 7.1: Descriptive statistics of the probability distributions used to validate our model, comparing modelled, simulated, and measured distributions.

Dist.	n	Modelled		Simulated		Measured	
		μ_A	$\eta_{S,A}$	μ_S	$\eta_{M,S}$	μ_M	$\eta_{A,M}$
B(40, 0.5)	2	1.090	0.01 %	1.090	0.86 %	1.099	0.86 %
	4	1.163	0.00 %	1.163	0.83 %	1.173	0.84 %
	8	1.225	0.02 %	1.225	0.85 %	1.236	0.88 %
	16	1.278	0.01 %	1.278	0.75 %	1.287	0.74 %
	32	1.325	0.01 %	1.325	0.11 %	1.326	0.12 %
Geo(0.05)	2	1.476	0.05 %	1.477	1.36 %	1.497	1.43 %
	4	2.047	0.03 %	2.047	0.91 %	2.065	0.89 %
	8	2.668	0.01 %	2.668	0.77 %	2.689	0.77 %
	16	3.317	0.02 %	3.317	0.12 %	3.321	0.14 %
	32	3.979	0.02 %	3.979	1.64 %	3.915	1.59 %
Pois(30)	2	1.104	0.00 %	1.104	0.57 %	1.110	0.58 %
	4	1.191	0.00 %	1.191	0.55 %	1.198	0.55 %
	8	1.268	0.01 %	1.268	0.57 %	1.275	0.57 %
	16	1.335	0.00 %	1.335	0.44 %	1.341	0.45 %
	32	1.397	0.00 %	1.397	0.26 %	1.393	0.25 %
U(20, 40)	2	1.118	0.00 %	1.118	0.57 %	1.124	0.58 %
	4	1.213	0.01 %	1.213	0.55 %	1.220	0.54 %
	8	1.275	0.01 %	1.275	0.58 %	1.282	0.59 %
	16	1.309	0.01 %	1.309	0.53 %	1.316	0.54 %
	32	1.326	0.00 %	1.326	0.16 %	1.328	0.16 %
NB(5, 0.3)	2	1.301	0.02 %	1.301	1.64 %	1.323	1.68 %
	4	1.587	0.06 %	1.586	1.38 %	1.608	1.34 %
	8	1.860	0.04 %	1.860	1.32 %	1.885	1.30 %
	16	2.123	0.06 %	2.124	0.84 %	2.142	0.90 %
	32	2.375	0.01 %	2.375	0.72 %	2.358	0.70 %

μ_A denotes the expected performance loss as derived analytically using our model, μ_S denotes the mean performance loss derived from our simulation, and μ_M denotes the mean of the measured data. $\eta_{a,b}$ denotes the relative error between the means μ_a and μ_b : $\eta_{a,b} = |(\mu_a - \mu_b)/\mu_a|$.

availability of sufficient device memory to store the input matrices.

As our GPU is of a post-*Volta* architecture, it is equipped with Independent Thread Scheduling (ITS) which implies that it is not strictly an SIMT device [272]. In order to more accurately simulate true SIMT behaviour, explicit thread group-level synchronisation² was added to the kernel. This also allows us to evaluate the effects of Independent Thread Scheduling by disabling this synchronisation, as explored in Section 7.5.4.

As discussed in Section 7.4.3, some of our underlying distributions had to be truncated to ensure finite support; in these cases, the acceptable loss of precision was set to a threshold of $\epsilon = 10^{-6}$.

7.5.3 Validation Results

In order to evaluate the quality of our model, we calculate the expected and mean performance loss for our modelled, simulated, and measured results. The nature of our problem makes it difficult to apply many of the usual goodness-of-fit tests; some of the probabilities we model are extremely small, leading to very small numbers of expected observations, which invalidates the use of Pearson's χ^2 and similar tests. Because our data is discrete, the Kolmogorov-Smirnov test (and other statistical tests for continuous distributions) are not applicable. This need not be a problem, however: as discussed in Section 7.4.3, we posit that the mean is one of the most meaningful statistics for our model, as it allows us to estimate the performance loss for entire workloads.

The results of our analysis are shown in Table 7.1. These results indicate that our model manages to predict the mean performance loss of both the Monte Carlo simulation and the measurements on the GPU with a high level of accuracy: the relative error between our model and the Monte Carlo simulation never exceeds 0.1 % in our validation, and the relative error between our model and the timing results from the GPU – a noisy environment – never exceeds 2 %. A visual comparison between the modelled, simulated, and measured results for a subset of distributions³ is shown in Figure 7.5. These figures confirm – on a visual level – our previous findings that the model agrees well with simulated and measured data.

We observe that finitely-supported distributions (namely, the binomial and uniform distributions) are modelled more accurately, due to the fact that they

²In CUDA terminology, this is referred to as warp-level synchronisation.

³The remaining distributions exhibit similar levels of agreement, but were not included due to space limitations.

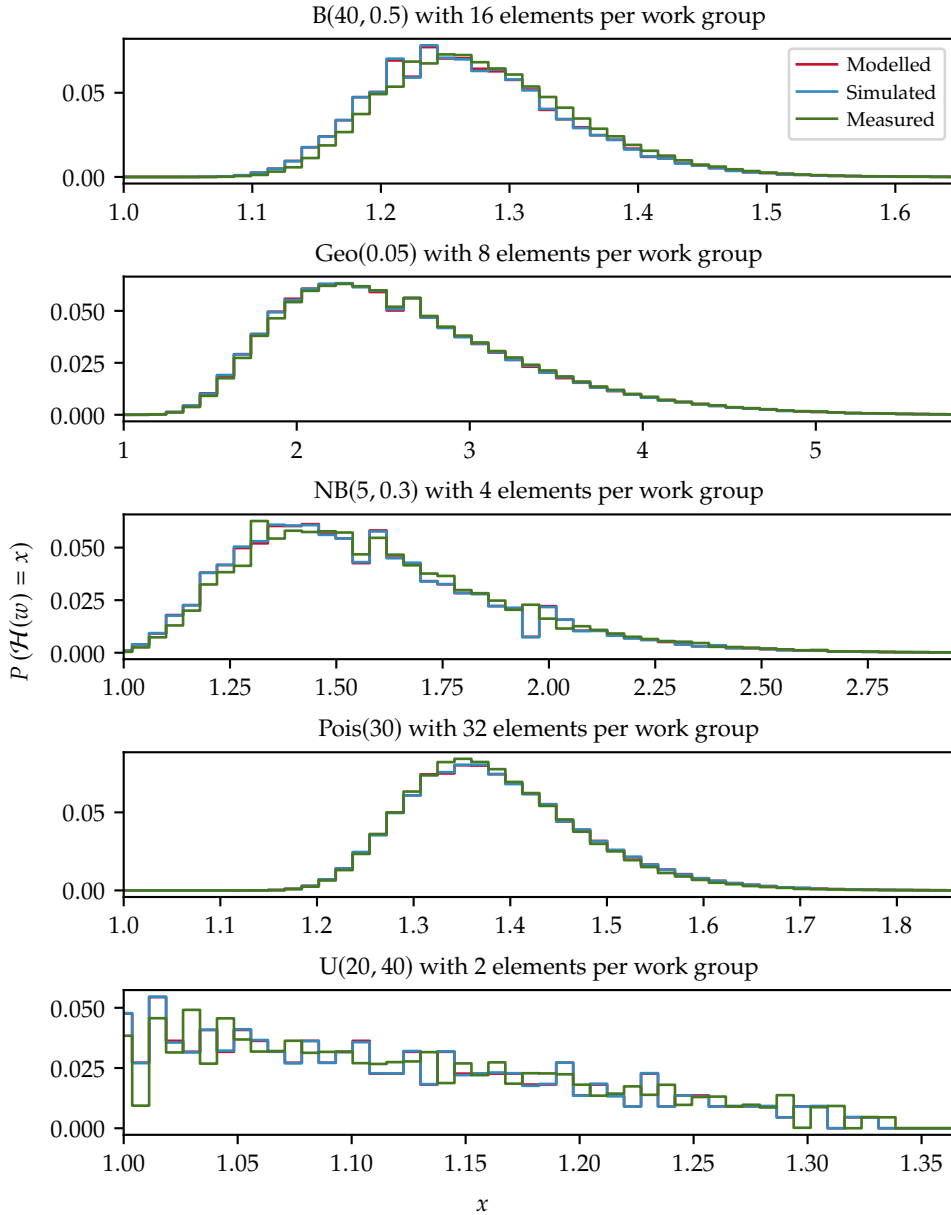


Figure 7.5: Comparison between the modelled, simulated, and empirically measured performance loss distributions for a subset of distributions and degrees of parallelism.

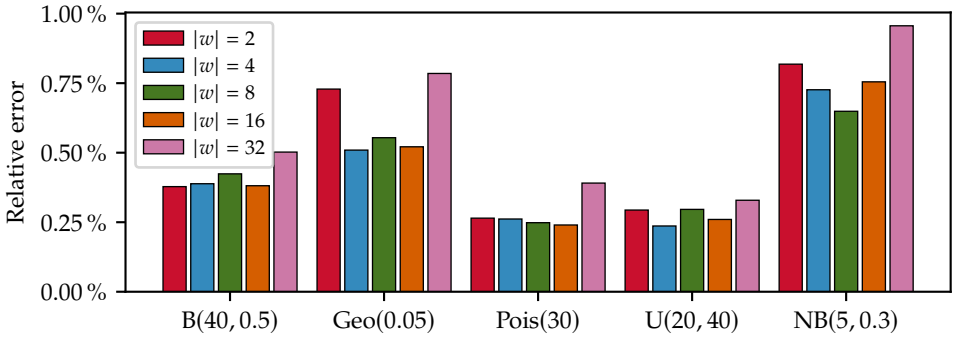


Figure 7.6: Relative error of measured performance loss with explicit thread group-level synchronisation enabled and disabled, demonstrating the impact of Independent Thread Scheduling.

do not require truncation; the truncation required to make our model work with infinite distributions (in this case, the geometric, Poisson, and negative binomial distributions) discards a small but non-zero amount of information.

7.5.4 Effects of Independent Thread Scheduling

Due to the Independent Thread Scheduling (ITS) architectural feature, the A100 GPU used in our validation is not a true SIMT device, as thread group-level synchronicity is not guaranteed by the hardware. As discussed in Section 7.5.2, we use explicit thread group synchronisation to more accurately emulate SIMT behaviour, but it remains prudent to investigate the effect of this non-SIMT architecture on our model. To this end, we have performed all our measurements with the explicit thread group synchronisation disabled and computed the difference in performance loss between the two sets of results. These differences are shown in Figure 7.6.

We observe that, in all of our benchmarks, the relative error between the experiments with and without explicit thread group-level synchronisation is less than 1%. We conclude that the presence of Independent Thread Scheduling has little impact on the accuracy of our model and, therefore, our model remains adequate even for future devices which may not follow the SIMT execution model in the strictest sense. It is worth noting that these results are consistent with the notion that ITS primarily serves to guarantee forward progress in parallel algorithms, rather than to provide significant gains in performance [273]. While we are not aware of any existing studies examining the effects of this microarchitectural fea-

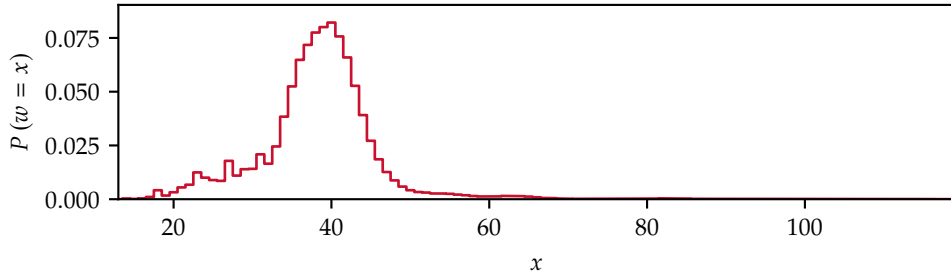


Figure 7.7: Distribution of the number of propagation steps, equivalent to the length of the work units, in our particle propagation mini-app.

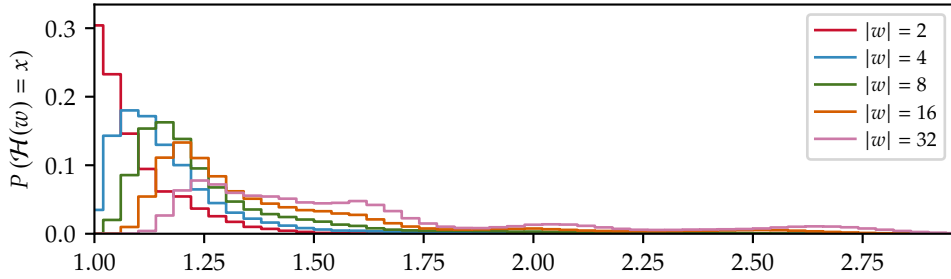


Figure 7.8: Distribution of modelled thread divergence imbalance for the distribution given in Figure 7.7.

ture in and of itself, our results are consistent with studies which – in passing – examine its impact on the performance of other applications [262, 274].

7.5.5 Limitations

The main limitation of our model is that the MIMD device which underpins it is inherently theoretical; however, we are not aware of any real-world pair of SIMT-MIMD processing devices with exactly equivalent computational power. This has consequences for the predictive power of our model, as we cannot use it to make concrete predictions about program run-time; instead, our model can be used – for example – to rank different implementations, as they are compared against the same theoretical optimum.

7.6 Practical Implications

Hitherto, we have considered our model only in the abstract and the synthetic, without consideration for its implications and application to performance modeling in real-world scenarios. To provide some insight into how this model may be applied in practice, we focus on one of the applications which inspired its inception: the propagation of particles in high-energy physics. As we recall from Chapter 4, the integration of the equations of motion for particles travelling through magnetic fields is an important computational problem, usually solved in a step-wise manner using an adaptive fourth-order Runge-Kutta-Nyström method [275].

A recurring question about particle propagation is which processing granularity is most efficient for execution on a GPU: one thread per particle, an entire thread group per particle, or some intermediate mapping. Answering this question is difficult because the number of propagation steps differs between particles: the application is subject to thread imbalance. The fact that thread imbalance is a significant hurdle in implementing Runge-Kutta-Nyström methods for SIMT devices is explored in depth by Murray [264]. In this section we demonstrate (in a simplified fashion) how our model may help solve such questions.

To understand how our model may help, we must understand the two different causes of overhead along the spectrum of possible mappings. The first cause of overhead is – as has been central to this chapter – thread imbalance: different particles will need different numbers of Runge-Kutta steps to reach their destination, and this variance in the size of the work causes overhead. We expect this overhead to be highest in a per-thread mapping, and we expect it to be zero at per-group granularity. Second, we incur overhead by spreading a single unit of work across multiple threads. In the case of particle propagation, for example, the Runge-Kutta sub-steps can be effectively executed in parallel, but the step size adjustment which follows is a purely sequential process. Amdahl’s law thereby dictates that we will achieve non-perfect speed-up due to the inherently sequential part of the problem [90]. In this second case, a per-thread configuration will incur no overhead, and a per-group configuration will be impacted the most. Thus, the optimal execution granularity is the mapping where these factors combine to give the lowest total overhead.

In order to construct a model for our particle propagation mini-app, we first measure the number of steps taken to perform each propagation using a propagation example provided by the *Acts* software package [122, 276]; the resulting distribution is shown in Figure 7.7. We then examine the cases of executing 1, 2, 4,

Table 7.2: Expected overhead due to thread imbalance $\mathbb{E}(X(n))$ and imperfect parallelism $h(n)$ (Equation 7.17), as well as the expected total overhead for given numbers of work units processed in parallel using our mini-app.

Parallel units	Threads per unit	$\mathbb{E}(X(n))$	$h(n)$	Total
1	32	1.000	2.476	2.476
2	16	1.104	1.714	1.893
4	8	1.193	1.333	1.591
8	4	1.300	1.143	1.486
16	2	1.447	1.048	1.516
32	1	1.647	1.000	1.647

8, 16, and 32 particles in parallel, which result in the overhead distributions shown in Figure 7.8. Furthermore, we assume that the execution time of the parallelisable Runge-Kutta sub-steps is $t_{\text{RK}} = 20$ units, and that the execution time of the sequential step size adjustment is $t_{\text{S}} = 1$ unit. We can then produce a simple – and equally abstract – model for the overhead due to the sequential parts of the problem, h , for n particles in parallel as a corollary of Amdahl’s law:

$$h(n) = \frac{t_{\text{RK}} + \left\lfloor \frac{32}{n} \right\rfloor t_{\text{S}}}{t_{\text{RK}} + t_{\text{S}}} \quad (7.17)$$

Finally, because both sources of overhead are modelled as ratios, we can multiplicatively combine the overhead of both sources in order to find the total expected relative overhead for each of our mappings. The results are given in Table 7.2 for the sake of example. It follows that the best-performing thread mapping (i.e. the mapping with the lowest combined overhead) would be to assign four threads per particle, or – equivalently – to process eight particles in parallel for each group of thirty-two threads.

We believe that this example serves to show that our model can give meaningful results with relatively little knowledge of implementation details. In this case, we required only knowledge about the underlying distribution of the iteration counts (which was gathered from a non-parallel implementation) and some timing results about simple operations (which could be derived through microbenchmarking).

7.7 Related Work

Performance models for SIMT devices – GPUs in particular – have been widely studied. A survey of existing modelling techniques, as well as a framework for classifying models, is given by Madougou, Varbanescu, Laat and Nieuwpoort [277]. Within their framework, the model presented in this chapter could be classified as a model for optimisation space exploration at a coarse abstraction level requiring zero knowledge about the hardware. While Madougou, Varbanescu, Laat and Nieuwpoort specify a class of modelling techniques described as *statistical*, this class does not accurately describes our model: rather than extracting the impact of the design space from a posteriori knowledge of execution time through data-centric, machine-learning based methods, we use statistical methods to make a priori predictions of a program’s performance, which fits in the category of analytical models. This categorisation would imply that the closest existing models to ours are models such as *Eiger* [278], *Stargazer* [279], and the model by Zhang, Hu, Li and Peng [280]. However, all these models require knowledge about the implementation of the SIMT program, which our model does not; therefore, we believe that our model can be applied much earlier in the application development process.

On the topic of thread imbalance, existing literature suggests that reducing it can lead to significant performance improvements, and that correctly modelling the overhead of divergence is a key factor in reducing it. Indeed, Bialas and Strzelecki [259] provide in-depth benchmarks of the effects of thread divergence on applications through empirical benchmarking. Hong, Kim, Oguntebi and Olukotun [263] propose methods for reducing imbalance in graph processing by mapping units of work with similar lengths onto the same thread groups. Khorasani, Gupta and Bhuyan [281] propose a method for balancing workloads within thread groups to eliminate thread imbalance. Frey, Reina and Ertl [267] examine methods for reducing thread divergence in iterative algorithms, which is the same class of algorithms studied in our work; we believe that the model described in our work may benefit such optimisations through more accurate predictions of overhead. Gautama and Gemund [282, 283] have designed models for thread imbalance similar to the model presented in this chapter, but their work targets different architectures which do not run in lockstep. Magni, Dubach and O’Boyle [284] discuss automatic thread-coarsening, which is closely related to the concept of granularity discussed in this chapter. While their approach is more generally applicable, they rely on machine learning methods rather than analytical

solutions. In a different paper, Magni, Dubach and O’Boyle [285] show that thread-coarsening can significantly speed up applications executed on SIMT devices; our model addresses this specific problem and is, therefore, complementary to their work.

7.8 Reproducibility and Reusability

The software developed to perform the modelling described in this chapter, as well as the data gathered and the scripts to visualise those data, are permanently archived on Zenodo [286], and have been made available at doi:10.5281/zenodo.10931331. Unfortunately, the MASCOTS’22 conference to which the work presented in this chapter was submitted did not feature an artifact evaluation track; as such, the artifact associated with this chapter was not formally peer reviewed. The software – named SMITE for *Statistical Model of Imbalance Thread Execution* – is, at the time of writing, available freely under the Unlicense on GitHub at <https://github.com/stephenswat/smite> [287].

7.9 Summary

In this chapter, we have answered Research Question 3 by presenting a model that gives insight into the performance of iterative applications with stochastic variable-length workloads on SIMT architectures such as GPUs. Using our model, we can estimate the performance loss that such applications incur due to thread imbalance. Our model is designed specifically to require as little a priori knowledge as possible, relying solely on an understanding of the statistical distribution of the amount of work that is to be processed by each thread. This information can be extracted from domain knowledge or from simulation through an existing implementation, thus requiring little to no information about the details of an SIMT implementation of the program, and allowing our model to be used in the early stages of application development. We show that our model is accurate within a relative error of 0.1 % compared to a Monte Carlo simulation, and within 2 % when compared to measurements on a real device. We believe our model can be used to quantitatively motivate and guide important optimisations of SIMT programs, in particular thread coarsening and load balancing.

8

Evaluating the Viability of Massively Parallel Track Reconstruction

*Hold on, I know you're scared,
but you're so close to heaven.
Eyes shut tight,
just pretend you're like a feather.*

— Laura Pergolizzi
(Musician)

To recap the contents of this thesis so far, we have provided a look at the state of the art in track reconstruction in Chapter 4. We have detailed the challenges that rear their heads when we try to implement track reconstruction on massively parallel systems, and we have developed methods and models to tackle those challenges – both in the context of track reconstruction and in massively parallel computing in general. It is now time to circle back to our target application and evaluate the viability of running track reconstruction on massively parallel architectures. It is time to look to the future, and to see where we stand in the face of the data deluge of the HL-LHC era.

The effort to develop track reconstruction software for massively parallel devices, including the work presented in this thesis as well as the effort of others, is consolidated in the TRACCC project [288]. The TRACCC project is a research and development programme within the ACTS project aimed at developing a full track reconstruction chain for heterogeneous hardware. A driving factor in the development of TRACCC has been the portability of code to different platforms, but we will primarily focus on its performance on CUDA-capable hardware, i.e. on NVIDIA GPGPU hardware. In this section, we will discuss the different implementations of algorithms in TRACCC and we will, wherever possible, evaluate their performance individually. Then, we will present a methodology of evaluating the performance of the heterogeneous track reconstruction chain as a whole in order to answer Research

Question 4, and we will conclude this chapter – and with it, this thesis – with an evaluation based on the aforementioned methodology in order to answer our Main Research Question.

This chapter is based on the following publications:

- Andreas Salzburger, Attila Krasznahorkay, Beomki Yeo, Joana Niermann and Stephen Nicholas Swatman. ‘Navigation, Field Integration and Track Parameter Transport Through Detectors Using GPUs and CPUs within the ACTS R&D Project’. In: *Proceedings of the 21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT’22)*. Oct. 2022
- Beomki Yeo et al. ‘traccc – GPU Track reconstruction demonstrator for HEP’. in: *Proceedings of the 7th International Connecting The Dots Workshop (CTD’22)*. May 2022
- Noemi Calace et al. ‘Seed Finding in the Acts Software Package: Algorithms and Optimizations’. In: *Proceedings of the 8th International Connecting The Dots Workshop (CTD’23)*. Oct. 2023
- Andreas Salzburger, Joana Niermann, Attila Krasznahorkay, Stephen Nicholas Swatman, Beomki Yeo and Guilherme Metelo Rita De Almeida. ‘traccc – A Close-to-Single-Source Track Reconstruction Demonstrator for CPU and GPU’. in: *Proceedings of the 26th International Conference on Computing in High Energy & Nuclear Physics (CHEP’23)*. May 2023

It is important to note that some of the implementations discussed in this section were developed in collaboration with others. The results presented in this section were gathered by the thesis author alone.

8.1 Preprocessing

We recall from Section 4.3 that the preprocessing step of track reconstruction involves the processing of raw detector data into spatially meaningful data from which tracks can be created.

8.1.1 Design and Implementation

As discussed in Section 4.3, the main objective of the preprocessing step in the track reconstruction workflow is to solve a Connected Component Analysis (CCA) problem on sparse data, with a density of approximately 1 % or lower. As noted in the aforementioned section of this thesis, CCL and CCA algorithms for such sparse

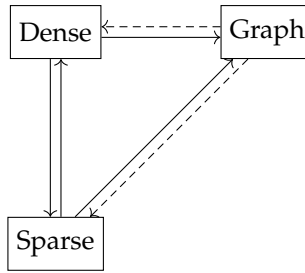


Figure 8.1: Reducibility of different flavours of connected component labelling problems. Solid arrows indicate reductions that are always possible, while dashed arrows indicate reductions that are possible if the input data has certain properties.

data are, themselves, sparse; i.e. there is relatively little research into how to solve such problems efficiently. To complicate matters further, algorithms designed for sparse datasets such as SPARSECCL [141] are strongly sequential, inhibiting our ability to implement them efficiently on GPGPU architectures. As such, we set out to find a method for solving sparse CCL problems that can be executed in a massively parallel fashion.

Our approach finds its root in the fact that there are, broadly speaking, three categories of connected component labelling problems. The first two categories are dense and sparse two-dimensional problems, and the third category concerns graph problems. We posit that these different CCL problems can be reduced to one another, as shown schematically in Figure 8.1. Dense problems can always be reduced to sparse problems by simply – but rather inefficiently – creating an equivalent sparse dataset. Similarly, sparse CCL problems can be reified into dense problems. Graph problems can be converted to dense or sparse problems if and only if the graph is planar, meaning that it can be laid out on a two-dimensional plane without any of the edges overlapping. Finally, we posit that sparse and dense CCL problems can always be converted into graph problems, and it is this reduction which we will exploit to efficiently implement the preprocessing stage of track reconstruction on massively parallel architectures.

The reduction of a dense CCL problem to a graph CCL problem is both trivial and efficient. Indeed, if we assume that we can access any coordinate (x, y) in the two-dimensional dense input in constant time, we can find all four or eight neighbours – in four-neighbourhood and eight-neighbourhood problems, respectively – in constant time, too. Converting each pixel in the input into a vertex in the graph and adding four or eight neighbouring edges gives a bounded lattice graph or

king's graph, respectively. Converting a sparse CCL problem to a graph problem follows a similar pattern, but suffers from the critical downside that constant-time accesses at arbitrary indices are no longer possible. Indeed, the sparse nature of such inputs makes it computationally costly to find the neighbours of a given cell using a naive approach, although performance can be significantly improved.

We recall from Section 4.3 that our sparse data is stored in COO format y -major, although the arguments made in this chapter hold for data stored with any major axis. We recall that for such data, the relation in Equation 8.1 holds between any two indices i and j in an input I .

$$\begin{aligned} \forall i, j < |I| : i < j \implies \pi_x(I(i)) < \pi_x(I(j)) \vee \\ (\pi_x(I(i)) = \pi_x(I(j)) \wedge \pi_y(I(i)) < \pi_y(I(j))) \end{aligned} \quad (8.1)$$

Equation 8.1 has two important corollaries. Firstly, we know that of the eight possible neighbours of an element at position i in an eight-connectivity problem, at most four have indices that are smaller than i , and four have indices that are greater than i . Furthermore, For every element at position i , there exist two other indices, $i_<$ and $i_>$ such that it is guaranteed that no elements connecting to i have indices smaller than $i_<$ or greater than $i_>$. To illustrate the existence of these elements, we first establish a binary relation, $R(i, j; I)$, which determines whether the element at index j is a valid eight-connective neighbour to i in an input I . This relation is given in Equation 8.2:

$$R(i, j; I) = |\pi_x(I(i)) - \pi_x(I(j))| \leq 1 \wedge |\pi_y(I(i)) - \pi_y(I(j))| \leq 1 \quad (8.2)$$

Intuitively, if there exists an element at index $i' < i$ such that $\pi_x(I(i')) = \pi_x(I(i)) - 1$ and $\pi_y(I(i')) = \pi_y(I(i)) - 1$, then $\forall j < i' : \neg R(i, j; I)$ and $i_< = i'$. Similarly, if there exists an element at index $i' > i$ such that $\pi_x(I(i')) = \pi_x(I(i)) + 1$ and $\pi_y(I(i')) = \pi_y(I(i)) + 1$, then $\forall j > i' : \neg R(i, j; I)$ and $i_> = i'$. If there do not exist any elements matching the aforementioned criteria, we can instead use the elements at the indices before and after the location at which these elements would be inserted according to the ordering defined in Equation 8.1. The argument presented here is equivalent to a more intuitive visual argument, which is that the eight possible neighbours of the element at index i form a square around that element, and this square has a top-left corner as well as a bottom-left corner. Any element that is

		$i' = 2$				
					$i' = 9$	
		$i_{<} = 3$			$i' = 10$	$i' = 13$
			$i = 5$	$i_{>} = 7$		$i' = 14$
	$i' = 1$	$i' = 4$	$i' = 6$			
$i' = 0$				$i' = 8$	$i' = 11$	
					$i' = 12$	

Figure 8.2: An example of a sparse image in y -major COO format. The central element at $i = 5$ is marked in red, and the cells in its eight-connectivity neighbourhood are marked in orange. Intuitively, no element with index $j < i_{<} = 3$ can be a neighbour of the central element, nor can any element with index $j > i_{>} = 7$.

either above the top-left corner of this square or on the same row but to the left of that corner can never be a possible neighbour of i , and neither can an element below the bottom-right corner or on the same row but to the right of that corner. This visual intuition is supported by Figure 8.2. An algorithm that is capable of efficiently finding the neighbours of cells in a sparse COO format is given in Algorithm 8.1.

The primary benefit of the aforementioned approach is that it allows us to efficiently convert a sparse CCL problem into a graph CCL problem. Graph CCL problems have – based on an examination of the available literature – been studied in much more detail than sparse CCL problems and, consequently, many more algorithms, including parallel algorithms, have been developed for graph CCL problems than for sparse CCL algorithms. In other words, the fact that we are able to efficiently reduce our existing problem to a graph CCL problem allows us to employ existing algorithms which have are known to be efficient in parallel environments. In the TRACCC project, we employ the FASTSV algorithm [293], an improvement on the classic Shiloach–Vishkin algorithm [294]. Although the exact details of our implementation are beyond the scope of this thesis, we find that the

Data: Sparse data I in y -major order, and an element i in I

Result: The set of outgoing edges for i , $E^+(i)$

```

 $E_{<}^+(i) \leftarrow \emptyset$ 
 $E_{>}^+(i) \leftarrow \emptyset$ 
 $(x, y) \leftarrow \pi_{x,y}(I(i))$ 
for  $i' \leftarrow i - 1$  to 0 do
     $(x', y') \leftarrow \pi_{x,y}(I(i'))$ 
    if  $x' < x - 1 \vee (x' = x - 1 \wedge y' < y - 1)$  then
        break;
    end
    else if  $|x - x'| \leq 1 \wedge |y - y'| \leq 1$  then
         $E_{<}^+(i) \leftarrow E_{<}^+(i) \cup \{i'\}$ 
        if  $|E_{<}^+(i)| = 4$  then
            break;
        end
    end
end
for  $i' \leftarrow i + 1$  to  $|I|$  do
     $(x', y') \leftarrow \pi_{x,y}(I(i'))$ 
    if  $x' > x + 1 \vee (x' = x + 1 \wedge y' > y + 1)$  then
        break;
    end
    else if  $|x - x'| \leq 1 \wedge |y - y'| \leq 1$  then
         $E_{>}^+(i) \leftarrow E_{>}^+(i) \cup \{i'\}$ 
        if  $|E_{>}^+(i)| = 4$  then
            break;
        end
    end
end
 $E^+(i) \leftarrow E_{<}^+(i) \cup E_{>}^+(i)$ 

```

Algorithm 8.1: Algorithm for efficiently finding the neighbours of a single element of a sparse image in y -major COO format. Performing this operation on each non-zero element returns a graph equivalent to the original sparse image.

FastSV algorithm is capable of solving our CCL problems quickly and efficiently.

8.1.2 Modelling Edge Case Behaviour

The FastSV algorithm, like many CCL algorithms, frequently updates a disjoint-set data structure which identifies the cluster to which each element in the input belongs. In order to significantly accelerate our implementation, we opt to store this data in fast, on-chip memory wherever possible. In NVIDIA parlance, this

memory is known as *shared* memory, whereas it is known as *local* memory in SYCL¹. In NVIDIA CUDA, shared memory is allocated on a per-block bases, which means that we must somehow find a partitioning scheme that allows us to break the input up into subsets, each suitable for processing by one block, i.e. anywhere between 32 and 1024 threads. Because communication between blocks is significantly more expensive than communication between threads in the same block, we aim to find a partitioning scheme that divides the input into completely disjunct sets between which communication – as defined by the eight-neighbourhood clustering scheme – is impossible. Thankfully, the nature of the input naturally provides such a scheme, as the input is divided across (for most experiments) thousands of separate detector surfaces; since one cluster cannot cross detector boundaries, we can simply partition across detector boundaries. In order to increase the regularity of the computation, we additionally allow our partitioning scheme to group together multiple detectors into a single partition if the total amount of data on those detectors is small².

Unfortunately, the aforementioned partitioning scheme has an important caveat. The shared memory on GPGPUs is – like all forms of memory³ – finite. For NVIDIA GPUs, the amount of shared memory is limited to 228 kB per Streaming Multiprocessor for the latest architectures, while older architectures can feature as little as 64 kB per Streaming Multiprocessor [97]. In our implementation of the FASTSV algorithm, the required amount of shared memory is linear in the size of the partition assigned to that block. It is possible, therefore, that a sufficiently large partition will require too much shared memory, leading to a crash in a naive implementation. We have opted to implement a system where, in the unlikely event that this happens, global memory can be used instead of shared memory, but global memory is much slower than shared memory and, more pressingly, our implementation only allows *one* block to use this global memory scratch space at the same time. Thus, it is useful to understand how likely it is for this edge case to

¹Shared memory, in NVIDIA vernacular, has little to do with what is known as shared memory in a broader parallel computing context. Traditionally, shared memory refers to memory that is accessible by all processors in a machine, while in NVIDIA jargon it refers to memory accessible by threads in a single thread block. Memory which is accessible by all threads on the GPU is known as global memory. To make matters worse, SYCL uses the term ‘shared memory’ to refer to memory that can be transparently transferred between processors without explicit memory transfers. This concept is referred to as ‘managed memory’ in NVIDIA parlance.

²We also investigated a more sophisticated scheme capable of splitting of detector modules across multiple blocks: if an entire row (or columns) of a detector is empty then, by the argument made in the previous subsection, eight-connectivity can never cross that row and it is a safe place to split. Unfortunately, most massively parallel programming models do not guarantee deterministic block scheduling, which would mean that measurements on the same detector could be discontinuous in memory, which is – for technical reasons – disadvantageous. Therefore, we chose to split the input on module boundaries only

³With the notable exception of the Turing machine’s infinite tape.

occur.

The probability that any given pixel is hit is the so-called *occupancy* of the detector, and is about 1 % for both the existing ATLAS pixel detector as well as the upcoming ATLAS ITk [295]. For a given module with n pixel, then, the chance that that module contains a given number hits is given by the binomial distribution $B(n, 0.01)$, and the chance that the number of hits on that module is lower than some limit m is thus given by $F_{B(n,0.01)}(m)$. Figure 8.3 illustrates the limit m (with which the amount of shared memory scales linearly) required for our implementation to achieve a 99 % per-event chance to not require the global memory scratch space at all. The aforementioned plot is based on the assumption that the current ATLAS Inner Detector is based on the FE-I4 readout chip⁴ with 2228 modules of $320 \times 144 = 46\,080$ pixels each [296, 297, 298], whereas the ATLAS ITk detector will feature approximately 9000 RD53 chips with $400 \times 384 = 153\,600$ pixels each [299]. As a consequence of these data, we currently aim for a value of $m = 2048$, as this provides a strong guarantee that use of the global scratch memory is unlikely. At a rate of 4 bytes per cell, this equates to about 8 KiB per block, allowing ample parallelism per Streaming Multiprocessor. Figure 8.4 shows the per-module probability of success for the ATLAS inner detector and the ATLAS ITk given $m = 2048$. We conclude that our implementation should fare well with data from the ATLAS ITk detector even on current GPUs.

8.2 Track Finding

The goal of the track finding process is to find sets of particle measurements that combine to form track candidates. The nature of track finding is strongly combinatorial and it is, as a result, one of the largest contributors to the running time of track reconstruction software. In this section, we investigate our implementations of massively parallel track finding; specifically, we describe implementations of its two constituent algorithms: seed finding and combinatorial Kálmán filtering.

8.2.1 Seed Finding

We recall from Section 4.4 that the seed finding stage of track finding aims to find triplets of spacepoints which form initial guesses to the particle trajectories in the event, which are later extended to tracks by the combinatorial Kálmán filter.

⁴In reality, the existing ATLAS inner detector is much more heterogeneous, but basing a model on the FE-I4 provides a pessimistic worst-case model.

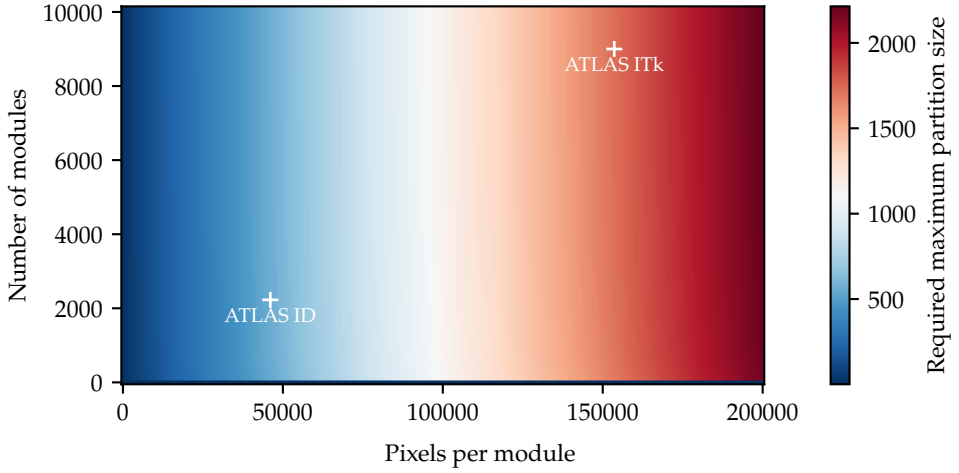


Figure 8.3: The required shared memory partition size in order to ensure that the per-event failure rate is 1 % for the ATLAS ID and the ATLAS ITk, assuming a hit density of 1 % for both detectors.

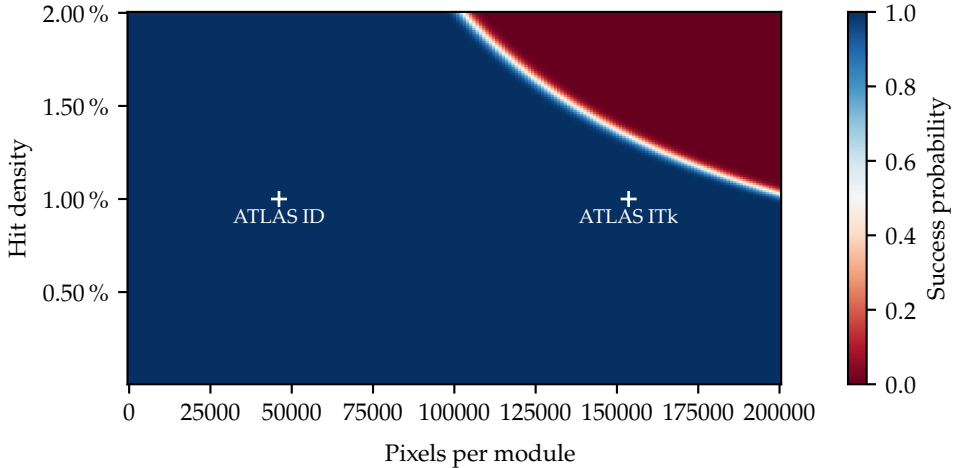


Figure 8.4: The per-module probability of running our clustering algorithm without the need for global-memory scratch space for the ATLAS ID and the ATLAS ITk, given a partition size limit of 2048.

In the TRACCC application, we propose two implementations of seed finding for massively parallel devices, which differ primarily in the way they find candidate spacepoints: one uses a predefined grid of spacepoints, while the other employs a k -d tree [300]. Both implementations are based on the idea of combining spacepoints into pairs, and then extending those pairs to triples as described in Section 4.4. In the remainder of this section, we will very briefly describe both implementations.

In the grid-based implementation, spacepoints are binned into a grid arranged across the ϕ , r , and z axes of the detector. For each spacepoint, pair candidates are then found by searching in a pre-defined number of adjacent bins. These pairs are then combined into triples. As discussed in Section 4.4, this approach technically increases the complexity of the algorithm from $O(n^3)$ – where n is the number of spacepoints – to $O(n^4)$, but the constant involved render it an efficient algorithm. Unfortunately, this grid-based implementation involves an unbounded number of spacepoints per bin, as well as an unbound number of pairs and triplets. In order to deal with these unbound and unknown variables, the grid-based seed finding algorithm is implemented using several kernels which count the size of the resulting dataset before another kernel writes the results to memory. Although this implementation results in highly efficient use of memory, it also performs a large amount of its computation twice in order for it to assess the size of the grids, as well as the size of the array of pairs and triplets found.

In the k -d tree-based implementation, on the other hand, the spacepoints are inserted into a three-dimensional k -d tree, partitioning the space in the ϕ , r , and z axes – similar to the grid used by the grid-based implementation. For a given spacepoint or pair of spacepoints, candidate extensions into, respectively, pairs and triples can be found by defining an axis-aligned search space, after which we can find matching spacepoints in $O(n^{\frac{2}{3}} + m)$ time, where n is the total number of spacepoints, and m is the number of points returned by the query [300]. Because an upper bound can be easily established on the size of a k -d tree – a k -d tree of n elements with a leaf node capacity of m will never contain more than $2^{n/m}$ leaf nodes and, as a result, never more than $n/m - 1$ nodes in total – no counting kernel is necessary to establish a geometric partitioning of the spacepoints.

Both approaches have up- and downsides which make it non-trivial to choose the best algorithm for a given use case. Indeed, the choice depends strongly on factors such as the geometry of the detector, the expected transverse momentum of the particles traversing that detector, and the expected occupancy. The grid-based implementation is much simpler, and allows us to express our requirements on spacepoint candidates directly. The k tree-based implementation, on the other

hand, requires us to first rewrite those requirements as axis-aligned search spaces; for some requirements, it may not be possible to precisely define an axis-aligned search space which requires additional filtering on individual spacepoints. On the contrary, environments which allow for tightly specified search spaces may see significant benefits from the use of k -d trees. In addition, the use of k -d trees decreases the number of unbound loop structures in the code which may lend performance benefits when executed on massively parallel hardware.

An implementation of the aforementioned algorithm employing k -d trees for seeding as previously implemented for traditional CPU-like devices in the ACTS project, where it has found success in some experiments where the search space can be highly constrained. In the remainder of this chapter, we will discuss the performance of the grid-based seed finding algorithm, as it is a more mature development.

8.2.2 Combinatorial Kálmán Filtering

We implement the combinatorial Kálmán filtering step of track finding using a loop of three kernels. This loop is based on the idea that a track candidate starts on a detector module and is then propagated to whichever detector the track intersects next; each iteration of the loop signifies the process of taking *all* tracks in the input and propagating them to the next detector. This loop is executed a bounded number of times depending on the detector: if a detector has n layers, then the combinatorial Kálmán filter will need to take roughly n steps. The three kernels in the loop each serve a distinct role in the process, which we will now detail briefly.

The first kernel – which we refer to as `apply_interaction` – models the increase in uncertainty as a particle travels through solid material. Indeed, detector modules have non-zero thickness and the sheer mass of this material (as well as, e.g. support beams) causes a particle to deviate from its original trajectory. As this interaction process is stochastic, we cannot be certain what its effect on the trajectory will be; we can only model it by adding uncertainty to the track parameters. For more information on material interactions, we refer to Section 4.4.3, but this kernel is not particularly interesting from a computational point of view as it simply maps a function over the input track parameters, and functions as a 1 : 1 association between input and output. Furthermore, the kernel contributes a negligible amount of execution time to our application, so we will not describe it in detail here.

The second kernel – `find_tracks` – is the kernel that introduces combinatorics into the Kálmán filter. This kernel assumes that the track has arrived at a surface, on

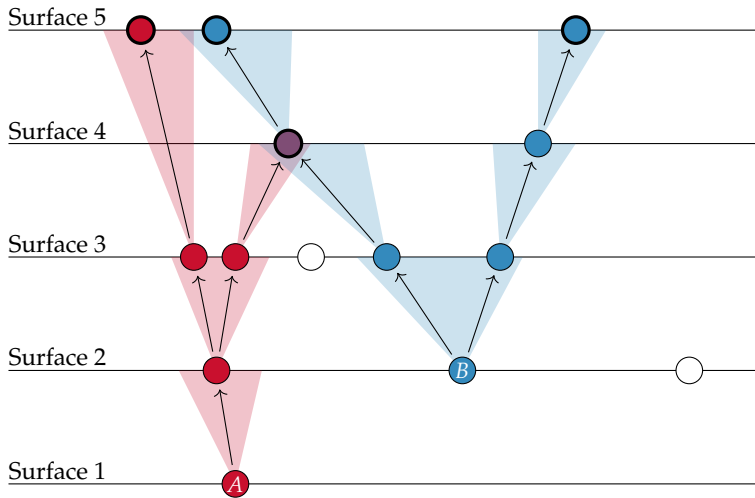


Figure 8.5: An example of the combinatorial Kálmán filtering; two tracks starting from *A* and *B* are extended across five surfaces. The propagation between surfaces (i.e. vertically, in this graphic) is performed by the `propagate` kernel, and the selection of measurements (i.e. the colouring of nodes on each layer) is done by the `find_tracks` kernel. Measurements with bold outlines are tips.

which zero or more measurements may be found. The track arrives at the surface at a given position and with a covariance matrix that describes the uncertainty – in the form of an ellipse – of that position. The `find_tracks` kernel then iterates over the measurements found within that ellipse and branches for each of them by performing a separate Kálmán update step, effectively creating one or more track candidates from a given initial track. Section 8.2.2 shows a schematic overview of the combinatorial Kálmán filter; the `find_tracks` kernel operates horizontally in this schematic, selecting the measurements that fall within the shaded uncertainty cones of the tracks. It is worth noting that it is possible for the `find_tracks` kernel to find zero measurements on a given surface. This naively indicates that the track doesn't match a real particle, i.e. that it is a fake track, as it violates the assumption that each particle leaves exactly one measurement on each surface that it passes. However, tracks that lack a measurement on a traversed surface – a so-called hole – are not necessarily fake: such situations can arise if a sensor is damaged due to long-term exposure to radiation or if the particle – by random chance – did not deposit sufficient energy to trigger the activation of the sensor. For this reason, we may choose to continue propagating tracks if zero measurements were found,

aborting the branch only if the number of holes becomes improbably large.

The third and final kernel, `propagate` does the numerical propagation of the track parameters through the detector volume using a forth-order Runge–Kutta–Nyström method. This involves numerical integration by stepping through the detector volume, as well as the updating of the covariance matrix to account for the additional uncertainty accrued while integrating. In Section 8.2.2, the `propagate` kernel operates vertically, moving from one surface to another. This propagation makes use of two principal components, known as the navigator and the stepper. The navigator keeps track of which surface will be the next to be intersected⁵ and is implemented in the `DETRAY` library [289, 301]. The `DETRAY` library is also responsible for representing the complex detector geometry – made up of many different shapes – in such a way that it can be efficiently accessed on a GPGPU. The stepper implements a Runge–Kutta–Nyström method and frequently accesses the magnetic field that permeates the detector, which is modelled using the `COVFIE` library described in Chapter 5.

At the time of writing, the `propagate` kernel – as well as the `fit` kernel – see Section 8.3 – are the largest contributors of latency in the `TRACCC` track reconstruction software. As shown in Figure 8.6, these kernels make up 75.4 % of the total kernel latency and they are, therefore, primary candidates for future optimisation. Interestingly, both these kernels rely strongly on propagation methods, i.e. the numerical integration of the equations of motion. As described in Section 4.4, such kernels suffer from inherent imbalance as the number of steps required to propagate from one surface to the next relies on a variety of factors including the homogeneity of the magnetic field and the physical layout of the detector. In the remainder of this section, we will investigate the performance of the `propagate` kernel to find avenues for future optimisation.

As discussed earlier, the combinatorial Kálmán filter is an iterative algorithm which requires roughly n iterations, where n is the number of layers in the detector. Figure 8.7 shows that the latency of kernels can vary significantly between iterations of the algorithm. Indeed, data suggest that it is primarily the early iterations that contribute significantly to the execution time, which correlates with the idea that the number of tracks increases in early iterations while it shrinks in later iterations. Intuitively, this follows from the fact that each iteration moves the track further from the centre of the detector and thus – in accordance with the inverse square

⁵In contrast to traditional ray tracing, particle tracks move non-linearly and – in heterogeneous magnetic fields – unpredictably; therefore, we cannot simply compute the next intersection. Instead, the next intersection must be constantly recomputed during the integration process

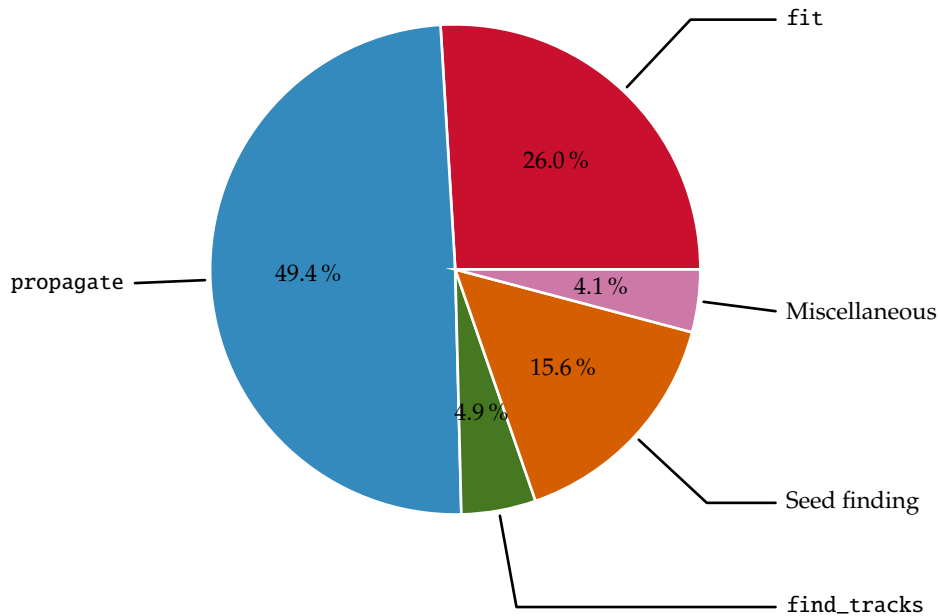


Figure 8.6: Pie plot of the relative contributions of kernel latency by different kernels or categories of kernels in the TRACCC track reconstruction software. For kernels which run multiple times per event, we sum the latencies of individual executions. Data taken from a $\langle\mu\rangle = 200\ t\bar{t}$ event in the Open Data Detector (ODD) [302] simulated using GEANT4 [303] and reconstructed on an NVIDIA RTX A5000 using a version of the TRACCC software dated October 16, 2024.

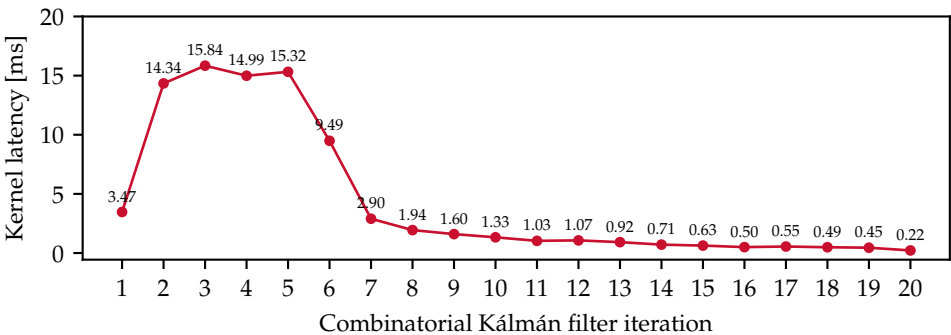


Figure 8.7: The latency of the propagate kernel in the twenty individual combinatorial Kálmán filter steps for a given event processed as described in Figure 8.6.

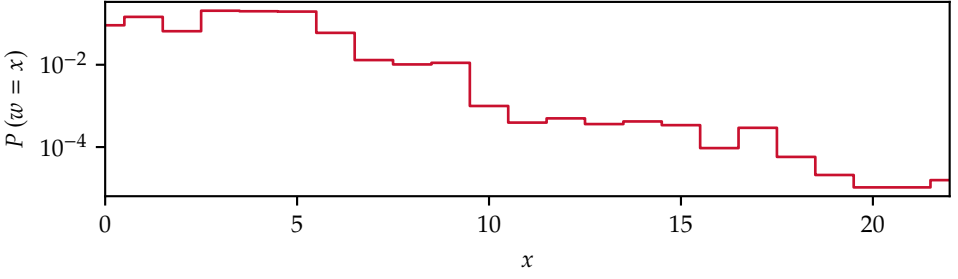


Figure 8.8: The distribution of the number of steps taken by individual threads in the propagate kernel in the second step of the combinatorial Kálmán filter for a given event processed as described in Figure 8.6.

law – the density of measurements decreases. Thus, the number of measurements that is compatible with each track shrinks and the branching factor drops. The fact that most of the kernel runtime originates from the first few iterations is expected, and as a result we will proceed to examine one of these early iterations. Specifically, the remainder of this section will examine the second iteration of the combinatorial Kálmán filter. These results were gathered using a $\langle \mu \rangle = 200 \, t\bar{t}$ event in the Open Data Detector (ODD) [302]. This event was simulated using GEANT4 [303] and reconstructed on an NVIDIA RTX A5000.

An examination of a kernel matching the aforementioned criteria using the NVIDIA NSIGHT COMPUTE profiler reveals several performance problems, including the fact that on average only 6.56 threads are active. Given the warp size of 32 threads on the NVIDIA RTX A5000 GPU, this means that we are using only 20.5% of the available threads, letting the remaining compute power go to waste. This problem arises primarily from imbalance in the number of propagation steps that individual threads make. Figure 8.8 shows the distribution of the number of steps per thread⁶; notable is the widely supported distribution, as well as the fact that many threads take exactly zero steps, i.e. they do not actually do any work at all. This pattern of computation matches the performance anti-patterns described in Chapter 7, and we will now apply the methods described in that chapter to better understand how to tackle the performance loss in the combinatorial Kálmán filter.

We recall from Chapter 7 that the metric $\mathcal{H}(w)$ describes the amount of performance lost due to imbalance, where w is a set of elements describing the amount of

⁶Astute readers may notice the similarities between the premise of Figure 8.8 and Figure 7.7; note, however, that these are taken from different software operating in different detectors. Although Figure 7.7 serves as a useful example of how the methods in Chapter 7 may be applied, Figure 8.8 pertains more strongly to the application at hand.

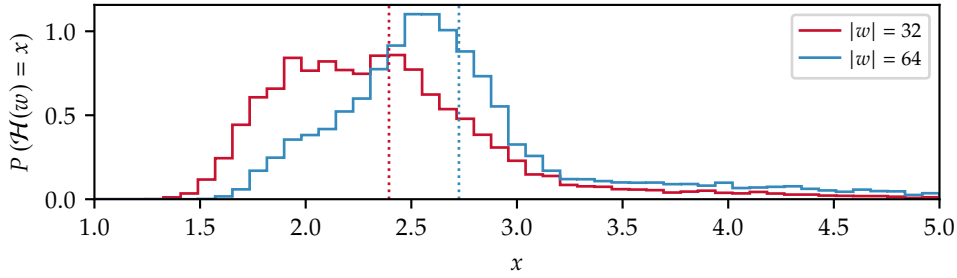


Figure 8.9: The modelled performance loss factor \mathcal{H} for warp sizes $|w| = 32$ (relevant for most NVIDIA GPGPUs) and $|w| = 64$ (some AMD GPGPUs) for the distribution given in Figure 8.8.

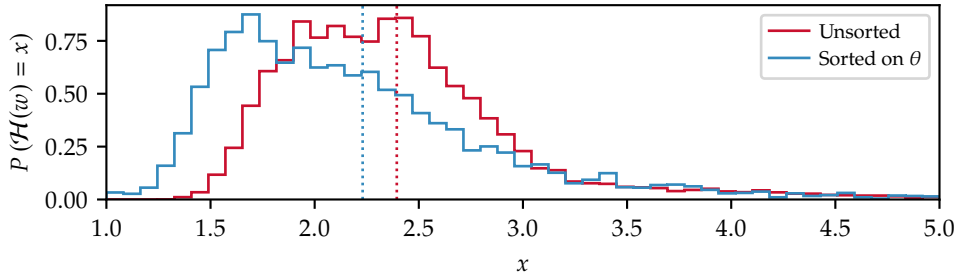


Figure 8.10: The modelled performance loss factor \mathcal{H} for a given event processed as described in Figure 8.6, with and without sorting of tracks by their angle along the beampipe θ .

work done by each thread, e.g. the number of propagation steps each thread. We also recall that $\mathcal{H}(w) \geq 1$, where a value $\mathcal{H}(w) = 1$ is optimal and a higher value indicates greater performance loss. Given the distribution in Figure 8.8 we obtain the distribution for $\mathcal{H}(w)$ shown in Figure 8.9. It is clear from the aforementioned plot that the amount of performance loss is significant for GPGPUs with warp size $|w| = 32$, and even more so for GPGPUs with warp size $|w| = 64$, which includes some AMD GPGPUs. In an attempt to decrease the performance degradation due to imbalance, we explored the possibility of sorting tracks by some measure that would correlate with the number of propagation steps. The angle θ which represents the angle between the particle and the beampipe was chosen; in brief, sorting tracks in this way ensures that tracks propagated by the same warp propagate through the same regions of the detector, meaning that they will be more likely to encounter the same detector geometry. Figure 8.10 indicates that this approach should, indeed increase performance. Empirical evidence supports this claim, and

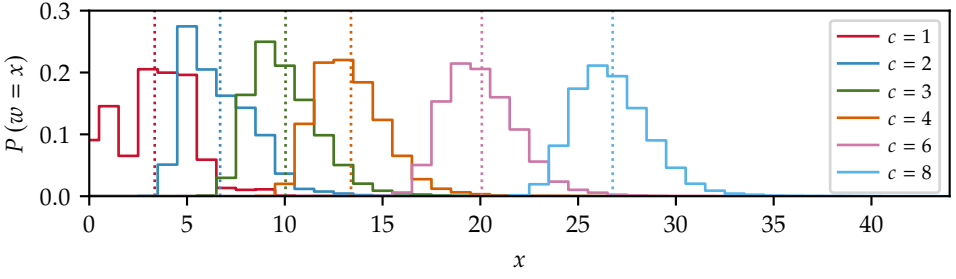


Figure 8.11: The distribution of work per thread given the distribution in Figure 8.8 given a thread coarsening factor c in a work-stealing regime. Note that the distribution for $c = 1$ is exactly equivalent to the distribution in Figure 8.8.

we observe an increase in the number of active threads by 25.6 %, from 6.56 to 8.24. Unfortunately, while this increase is significant it still leaves a very low thread utilisation of 25.7 %.

As an alternative to a simple sorting strategy, we advocate for a thread coarsening approach. In other words, we advocate for allocating more than a single tracks to each individual thread and allowing a work-stealing strategy to balance the workload between threads. We characterise the degree of thread coarsening by a factor $c \in \mathbb{N}^+$, where the number of tracks assigned to each warp is $c|w|$; in other words, each thread processes – on average – c threads, and the existing approach is characterised exactly by $c = 1$. We expect that increasing c will increase the amount of work per thread, while the distribution should – by the central limit theorem – approach a normal distribution. The results of a Monte Carlo simulation of work stealing, given in Figure 8.11, support these ideas; the mean workload per thread scales exactly with c – indeed, work does not magically disappear – while the distribution moves further away from zero. The important consequence of the latter is that it significantly decreases the ratio between the expected maximum workload in a given warp and the mean workload, thereby decreasing $\mathcal{H}(w)$.

Computing the expected performance loss factors for the distributions in Figure 8.11 gives Figure 8.12; it is clear from these data that thread coarsening does indeed – in a theoretical model – drastically reduce the performance loss due to imbalance. It is worth noting that there are potential downsides to thread coarsening, namely the reduction of the degree of hardware utilisation; fortunately, the existing non-coarsened implementation issues – on average for a $\langle \mu \rangle = 200$ event – eleven full wavefronts of threads, so there is sufficient room for us to decrease

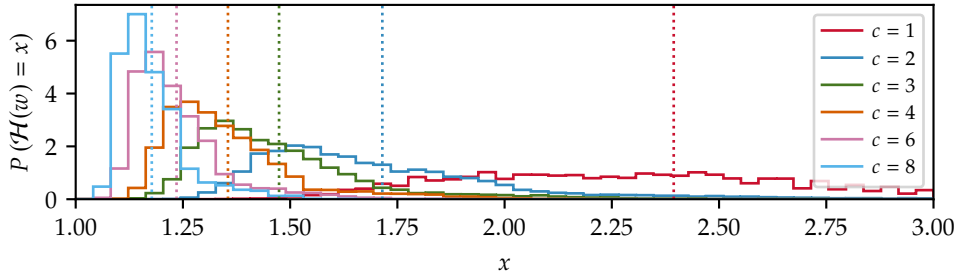


Figure 8.12: The predicted performance loss metric $\mathcal{H}(w)$ for the coarsened distributions given in Figure 8.11.

the kernel grid sizes (through coarsening) before occupancy becomes a significant issue. In summary, we believe that thread coarsening is the obvious way forward to improve the performance of the combinatorial Kálmán filter and the performance of our massively parallel track reconstruction application as a whole.

8.3 Track Refinement

In addition to the preprocessing and track finding steps, we also implemented a massively parallel track fitting algorithm based on the Kálmán filter as described in Section 4.5.1. For the sake of brevity, however, we will not cover the implementation of this algorithm in detail. Indeed, the Kálmán filter-based track fitting algorithm is – from a computational point of view – strictly simpler than the combinatorial Kálmán filter described in the previous section. The Kálmán filter-based track fitter does not have a combinatorial component, which is one of the primary complications of the combinatorial Kálmán filter. Given the non-combinatorial nature of the fitting algorithm, it need not be composed into different kernels as the input never grows and we never need, therefore, to allocate larger memory buffers. It is worth noting that the fitting algorithm employs a more complex Kálmán update step in order to perform smoothing, but the fact that this computation can be executed in an embarrassingly parallel fashion renders it largely uninteresting from the perspective of parallel processing. Our implementation of a massively parallel Kálmán filter-based track fitting algorithm shares many similarities with our implementation of the combinatorial Kálmán filter track finding algorithm, including the use of the DETRAY library for storing detector data, and the COVFIE library (see Chapter 5) for representing magnetic fields.

Notably, the track fitting algorithm implemented remains a large contributor

of the total runtime; as Figure 8.6 shows, the single invocation of the `fit` kernel takes over half the time of all the iterations of the `propagate` kernel described in the previous section. Thankfully, the reason why the fitting algorithm takes such a relatively large amount of time is similar to the reason why the `propagate` kernel does: it exhibits a very similar propagation pattern, and it suffers from the very same imbalance that the `propagate` does. As such, we believe that the optimisation strategy expounded in the previous section will also apply to the fitting algorithm. The implementations of such a thread coarsening strategy in the track fitting algorithm remains – as it does in the combinatorial Kálmán filter – future work.

8.4 Task Graph Performance Bounds

We have now explored our implementations of novel algorithms for various parts of the track reconstruction workflow, but we have yet to evaluate its performance as a whole. Holistically evaluating that performance is a daunting task, as the performance of the application as whole depends strongly on the scheduling and placement of individual algorithms. Finding good scheduling and placement strategies remains an open challenge in high-energy physics scheduling software such as Gaudi [304]. In lieu of an environment in which scheduling can be easily and efficiently implemented, we employ a theoretical model of upper bounds on performance in order to evaluate our massively parallel track finding software. In this section, we describe our approach.

8.4.1 Background

Theoretical models of task graphs – including in heterogeneous systems – have been extensively studied. This comes as no surprise, as task graphs are broadly applied and models of performance can be powerful predictive and descriptive tools. Castrillon, Desnos, Goens and Menard have modelled task graph applications on heterogeneous system-on-chip architectures in a way that is very similar to the approach described in this chapter [305]. Rehn-Sonigo study both the mapping and scheduling of task graphs onto heterogeneous platforms, employing models of such computations to evaluate different approaches against one another [306]. Finally, Kao, Krishnamachari, Ra and Bai study the modelling of latency in task graph applications in order to facilitate the minimisation of overhead in latency-critical systems [307].

Many existing models – and scheduling methods – for heterogeneous systems are based on linear programs. Linear programs are mathematical optimisation problems which aim to maximise – or minimise – a linear combination of values, subject to a set of constraints. Canonically, linear programs are expressed as in Equation 8.3, where \vec{x} is the solution vector, \vec{c} weights the solution vector, and A and \vec{b} provide the constraints on \vec{x} [308]:

$$\begin{aligned}
 &\text{find} && \vec{x} \\
 &\text{that maximises} && \vec{c}^\top \vec{x} \\
 &\text{subject to} && A\vec{x} \leq \vec{b} \\
 &&& \vec{x} \geq 0
 \end{aligned} \tag{8.3}$$

Examples of applications of linear programs in the field of scheduling include the work by Cuervo et al. who study the offloading of computation from low-power mobile devices [309]. Boiniski and Czarnul specifically study the assignment of separable computations onto multiple devices using linear programming [310]. Finally, Benoit, Dobrila, Nicod and Philippe study the use of linear programs to understand the mapping of computation onto hardware that is itself reprogrammable, such as FPGAs [311]. All these approaches employ integer linear programming to find optimal schedules and mappings for heterogeneous systems. In the remainder of this section, we describe a methodology which is, in many ways, much simpler: we employ real-valued linear programs not to provide an optimal scheduling, but rather to provide upper bounds on task graph performance.

8.4.2 Optimistic Throughput Modelling

In order to assess the performance of the hitherto presented reconstruction software, we will now present an optimistic upper bound on the achievable throughput of the track reconstruction pipeline on heterogeneous systems. Such an upper bound will allow us to quantify how much performance remains to be gained, and a generalisable method of generating such upper bounds for different system configurations will allow us to make additional predictions about the efficacy of adding additional processing units to a system. In short, a throughput model for our application is a powerful tool both for assessing the work presented in this thesis so far, as well as for guiding future work.

Since this thesis is concerned with an application in the form of a task graph, we propose a method for modelling throughput that specifically incorporates the

structural nature of such graphs: we propose to represent them as *flow networks*. In general, a flow network consists of a set of nodes V , a set of edges E , a flow limit per edge $f : E \rightarrow \mathbb{R}^+$, and a source and sink node $s, t \in V$. The flow value for an edge is some abstract representation of the ability of that edge to process flow: in real world terms, this could be the number of litres of water that can flow through a given river in a certain amount of time or the number of cargo containers that can be carried on a given railroad. In computational terms this could, for example, be the amount of data processed by a given algorithm. The objective of the maximum flow problem is to find an allocation of flow to edges such that no edge carries more than its limit, while also maximising the amount of flow between source s and sink t . In this model, the source s represents the input data type of the application, and the sink t represents the desired output data type.

To see how flow networks can be used to model task graphs, consider an application consisting of three types forming the set $T = \{A, B, C\}$ and two kernels between those types forming the set $K = \{k_1 : A \rightarrow B, k_2 : B \rightarrow C\}$. These kernels can be run on two heterogeneous devices forming the set $D = \{d_1, d_2\}$. The vertices in our flow network are then given by the Cartesian product of the different data types T and the devices D in our task graph. We use $d(Q)$ to refer to the type $Q \in T$ residing in the memory of device $d \in D$. In the aforementioned example, our flow network would contain six data vertices $V = \{d(Q) : d \in D, Q \in T\} = \{d_1(A), d_2(A), d_1(B), d_2(B), d_1(C), d_2(C)\}$. The edges E in our flow network are defined as being of one of two kinds. The first kind of edge is the set of kernel edges, i.e. the set of kernels bound to a specific device. Much like how the set of vertices V is the product of the set of devices D and the set of types T , the set of kernel edges is given as the product of the set of devices D and the set of kernels K , such that $E_K = \{d(k) : d \in D, k \in K\}$ where $d(k)$ denotes the execution of kernel k on device d . Naturally, the input and output types of $d(k)$ are equivalent to the input and output types of k , except that they are stored on device d , i.e. $d(k) : d(\text{dom}(k)) \rightarrow d(\text{codom}(k))$. For our working example, we obtain $E_K = \{d_1(k_1) : d_1(A) \rightarrow d_1(B), d_1(k_2) : d_1(B) \rightarrow d_1(C), d_2(k_1) : d_2(A) \rightarrow d_2(B), d_2(k_2) : d_2(B) \rightarrow d_2(C)\}$. Finally, our flow network must contain edges E_I to represent the ability to transfer data from one device to another. We define $E_I = \{(d_1 \triangleright d_2)(Q) : d_1, d_2 \in D, Q \in T\}$ where $(d_1 \triangleright d_2)(Q) : d_1(Q) \rightarrow d_2(Q)$ denotes the transfer of type Q from device d_1 to device d_2 . Our example graph contains six such edges. The set of edges E is the union of $E_K \cup E_I$. A graphical representation of the flow network described in this paragraph is shown in Figure 8.13.

To complete our flow network, each edge must be given a flow limit $f : E \rightarrow \mathbb{R}^+$.

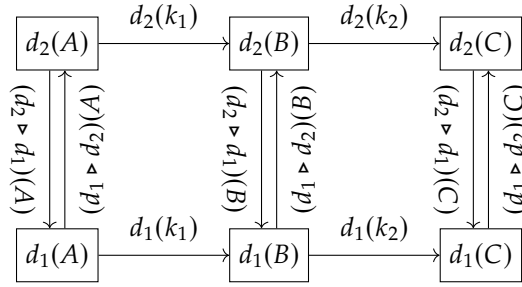


Figure 8.13: An example of a flow network given by two devices, $D = \{d_1, d_2\}$, and two kernels, $K = \{k_1 : A \rightarrow B, k_2 : B \rightarrow C\}$.

In the case of kernel edges, e.g. $d(k) \in E_K$, we define the flow limit of that edge as the number of times the corresponding device, i.e. d , can execute the corresponding kernel, i.e. k , in a given unit of time. The throughput of kernels on different devices is dependent on a variety of factors, including but not limited to the arithmetic throughput of that device, the throughput of its memory, and the efficiency of the implementation. For edges representing interconnects between devices such as $(d_1 \triangleright d_2)(Q) \in E_I$, we define the flow limit as the number of objects of the corresponding type, e.g. Q , can be transferred from the first device to the second device, e.g. from d_1 to d_2 . This can be conveniently calculated as the quotient of the bandwidth of the interconnect and the size of the data type. Note that for any edge $e \in E$, $f(e) = 0$ indicates that either a given kernel cannot be executed on a given device, or that a given data type cannot be transferred between two devices. With the definition of these flow limits, the construction of our flow network is now complete.

With a sufficient amount of suspension of disbelief, solving the maximum flow problem on the aforementioned flow network from a given source node (say, $d_1(A)$, if we assume that d_1 is a CPU-like device which coordinates computation) to a given sink node (say, $d_1(C)$) now gives us a model of the achievable throughput of the task graph as a whole. Unfortunately, this model suffers from unrealistic assumptions about the task graph and the devices on which it is computed. The first assumption is somewhat subtle and fundamental to the nature of the proposed model, and states that the transfer of and the computation on data can be viewed as continuous processes. In the real world, the scheduling of data transfer and computation is as difficult and as widely studied as it is precisely because data and computation are *discrete*. A model which assumes continuous, infinitesimally

divisible units of computation, therefore, models a perfect static schedule which we are unlikely to find in practice. It is from this simplification that our model derives its optimistic nature.

Secondly and more obviously, the aforementioned model is naive to the fact that edges in the flow network contend for resources. Indeed, Figure 8.13 has two edges ($d_2(k_1)$ and $d_2(k_1)$) which compete for the computational ability of device d_2 . Similarly, two edges compete for computation on d_1 , and six edges compete for the full-duplex interconnect between d_1 and d_2 . Since a maximum flow algorithm has no way to consider these restrictions, naively finding the maximum flow in the aforementioned model using a maximum flow algorithm such as the Ford–Fulkerson algorithm allows for arbitrary overallocation of resources – both processors and interconnects – which is incompatible with real-world behaviour. In order to resolve this problem, we utilise the fact that maximum flow problems can be reduced to linear programs [308] which will allow us to add resource constraints to our model.

It is well known that maximum flow problems can be represented as linear programming problems [308], and that this reduction is efficient for real-valued flow networks. Indeed, the maximum flow problem is solvable in polynomial time [312], and linear programming is solvable in weakly polynomial time [313]. It is interesting to note that linear *integer programming*, on the contrary, is NP-complete [313] – this step-up in complexity when going from continuous data to discrete data is strikingly similar to the difference between scheduling discrete computation compared to solving a flow problem for continuous computation and data transfer.

A general recipe for transforming a maximum flow problem into a linear program is as follows. Firstly, we define a flow variable \vec{x}_e for each edge $e \in E$. Then, we define our goal as the maximisation of the flow over the sum of nodes coming from the source node or, equivalently, flowing into the sink node. We denote the set of incoming edges for a vertex $v \in V$ as $E^-(v)$, and we denote the set of outgoing edges as $E^+(v)$. By construction of the flow network, we find $E^-(d(Q)) = \{d(k) : k \in K, \text{codom}(k) = Q\} \cup \{(d' \triangleright d)(Q) : d' \in D\}$ and $E^+(d(Q)) = \{d(k) : k \in K, \text{dom}(k) = Q\} \cup \{(d \triangleright d')(T) : d' \in D\}$. Then, we add constraints for each edge so that the flow through that edge is positive and no higher than the flow limit of that node, i.e. $\forall e \in E : 0 \leq \vec{x}_e \leq f(e)$. Finally, we have to ensure that for all nodes except the source node s and sink node t , the incoming flow is equal to the outgoing flow, such that $\forall v \in V \setminus \{s, t\} : \sum_{e \in E^+(v)} \vec{x}_e = \sum_{e \in E^-(v)} \vec{x}_e$. This construction gives rise to the linear program given in Equation 8.4:

$$\begin{aligned}
 &\text{find} && \vec{x} \\
 &\text{that maximises} && \sum_{e \in E^-(t)} \vec{x}_e \\
 &\text{subject to} && \forall e \in E : 0 \leq \vec{x}_e \leq f(e) \\
 &&& \forall v \in V \setminus \{s, t\} : \sum_{e \in E^+(v)} \vec{x}_e = \sum_{e \in E^-(v)} \vec{x}_e
 \end{aligned} \tag{8.4}$$

It is from a linear program of this form that we can tackle the issue of resource overutilisation. To reiterate, the problem is that the constraints $\forall e \in E : 0 \leq \vec{x}_e \leq f(e)$ allows each of multiple edges to fully utilise a single resource, given that $f(d_i(k_j))$ is the maximum throughput of a kernel k_j if it is able to fully utilise device d_i , and that there may be multiple edges utilising the same device. Similarly, multiple edges may simultaneously fully utilise an interconnect which is impossible in the real world. In order to resolve this issue, we discard the idea that the variable \vec{x}_e represents the total flow through edge e in favour of the idea that \vec{x}_e represents the relative (defined on the range $[0, 1]$) usage of the resource – either a device or an interconnect – used by edge e . The flow through an edge e can then be computed as $\vec{x}_e f(e)$. This change allows us to add additional constraints to ensure that the total utilisation of each device is no greater than one. This, in addition to a constraint that none of the utilisation should be negative, gives rise to the updated linear program in Equation 8.5:

$$\begin{aligned}
 &\text{find} && \vec{x} \\
 &\text{that maximises} && \sum_{e \in E^-(t)} \vec{x}_e f(e) \\
 &\text{subject to} && \forall e \in E : 0 \leq \vec{x}_e \leq 1 \\
 &&& \forall v \in V \setminus \{s, t\} : \sum_{e \in E^+(v)} \vec{x}_e f(e) = \sum_{e \in E^-(v)} \vec{x}_e f(e) \\
 &&& \forall d \in D : 0 \leq \sum_{e \in \{d(k) : k \in K\} \cap E_K} \vec{x}_e \leq 1 \\
 &&& \forall d_1, d_2 \in D : 0 \leq \sum_{e \in (\{(d_1 \triangleright d_2)(Q) : Q \in T\} \cup \{(d_2 \triangleright d_1)(Q) : Q \in T\}) \cap E_I} \vec{x}_e \leq 1
 \end{aligned} \tag{8.5}$$

We posit that the model presented in this section can be used to evaluate the performance of an existing task graph implementation against an upper bound. Indeed, since the primary simplification made in our model concerns scheduling, our model allows for the evaluation of an existing task graph against a theoretical case in which an optimal scheduling exists. This use case is particularly interesting to us, as scheduling remains an open problem in our high-energy physics application. Additionally, our model can be used to guide future optimisation; akin to how

```

1 datatypes:
2   A: 128 # Size in bytes
3   B: 64
4   C: 1024
5 devices:
6   - d1
7   - d2
8 interconnects:
9   - source: d1
10     destination: d2
11     bandwidth: 100
12     bidirectional: true
13 algorithms:
14   k1:
15     in_type: A
16     out_type: B
17     implementations:
18       - device: d1
19         throughput: 5000
20       - device: d2
21         throughput: 5
22   k2:
23     in_type: B
24     out_type: C
25     implementations:
26       - device: d1
27         throughput: 10000
28       - device: d2
29         throughput: 10
30 source: A
31 sink: C\end{lstlisting}

```

Listing 8.1: An example of a program specification that closely matches the heterogeneous task graph program shown in Figure 8.13.

Amdahl’s law dictates that speed-up in one part of an application does not lead to equivalent speed-up in the application as a whole, the same goes for task graphs. In fact, such effects are even more complex in task graph applications because, e.g. reducing the execution time of one kernel not only makes that kernel run at a higher throughput, it also frees up computational resources for other kernels. Our model makes it trivial to quickly evaluate possible optimisation strategies through differential performance analysis, i.e. it can answer questions of the form ‘how does the performance of an application change if one of its kernels is sped up by a given factor’?

In order to facilitate the use of the model presented in this paragraph, we have written a Python-based program that ingests YAML-based program specification such as the example given in Listing 8.1 and returns an upper bound on the throughput of that program; more information on our implementation is given in Section 8.6.

8.4.3 Modelling Individual Kernel Throughput

The measurement of throughput for individual kernels presents an additional challenge to the model presented in the previous section, as the parameters for our model are the throughputs of kernels when the corresponding computational resource is fully utilised. In practice, however, very few kernels in the application under study are capable of truly exploiting the full computational prowess of the GPUs on which they run. Therefore, the throughput of these kernels cannot be determined naively by measuring their runtime. Instead, we opt to model the throughput of our kernels using a variety of easily computable metrics. In this section, we will present the model that we will use to compute the theoretical throughput of GPU kernels, which we will subsequently use in the model of the throughput of our application as a whole presented in Section 8.4.2.

Core to our model for kernel throughput is the idea that throughput of a kernel – or, more broadly, of any process which does not have dependencies on previous invocations of itself – can be computed as the number of processes N that can be performed simultaneously divided by the latency L of each individual process, which gives rise to Equation 8.6:

$$T(N, L) = \frac{N}{L} \quad (8.6)$$

Equation 8.6 is intuitive, as are some of its consequences: increasing the number of processes that can run simultaneously by a given factor will increase throughput by the same factor. Similarly, decreasing latency by a given factor will also increase throughput by the same factor. Another useful property of Equation 8.6 is that it can be easily computed for most kernels. The latency parameter L can be directly measured by any of the myriad profiling tools available for GPU applications or even – for those with a preference for more direct methods – through the use of timing functions. The number of simultaneous kernel executions is more difficult to compute, but we will proceed to show that this parameter, N , can also be easily computed for most kernels.

A naive computation of the number of concurrent executions for a given kernel follows from the number of threads required by the kernel, c_r , and the number of cores available on the device, c_d . For example, we may use the ratio $N = c_d/c_r$ as an approximation for how many instances of a kernel can be executed simultaneously, but this approach suffers from two main problems; the first pertains to the difference between threads, cores, and thread slots, as well as the degree to which those resources can be occupied. The second issue pertains to so-called *tail effects*.

Table 8.1: Overview of the number of thread slots per Streaming Multiprocessor (SM) for various Compute Capability (CC) levels in NVIDIA GPGPUs [97].

CC	Slots per SM
1.0 – 1.1	768
1.2 – 1.3	1024
2.0 – 2.1	1536
3.0 – 7.2	2048
7.5	1024
8.0	2048
8.6 – 8.9	1536
9.0	2048

In both cases, the core idea is that the latency metric L refers to wall-clock time and thus, intuitively, incorporates any effects that might slow down execution and, for Equation 8.6 to make sense, the parallelism metric N must be aware of the same issues. We will dedicate the coming paragraphs to more detailed explanations of these problems, and their possible solutions.

Firstly, the mapping of software threads onto hardware differs significantly between traditional CPU-like architectures and massively parallel GPGPU architectures. Comparatively, the design of CPU architectures is relatively simple: the number of software threads is unbound, and the number of cores is fixed. On such architectures, threads are scheduled onto cores through e.g. a FIFO or round-robin scheme, in arbitrary order. Furthermore, these threads may be context-switched – either voluntarily or through preemption – and control may be therefore be yielded to other threads in arbitrary order. On GPGPU architectures, this mapping is much more complex in order to facilitate a different model of context switching. CPU architectures context switching by transferring the complete state of a thread to memory such that it can be restored when control is returned to the thread. In order to reduce the overhead of context-switching, GPGPU architectures implement a zero-cost context-switching strategy that works by 1. implementing hardware thread slots, of which there are multiple per core; by 2. ensuring that the resources required by each thread are allocated for the entirety of the lifetime of that thread; and by 3. ensuring that a thread can only vacate a thread slot by completing execution. Given this design, the aforementioned computation of $N = c_d/c_r$ – where c_d is

the number of cores on the device – is inappropriate⁷. Rather, we should use the number of thread slots, which we will denote c_s . Thus, we obtain $N = c_s/c_r$. The values of c_s can be easily obtained by multiplying the number of compute units – *Streaming Multiprocessors* in NVIDIA parlance – by the number of thread slots per compute unit. The latter number depends on the *Compute Capability* of the device⁸ and is easily obtained from the NVIDIA CUDA documentation and is also given in Table 8.1 for posterity.

As an additional complication of the zero-cost context-switching model employed by GPGPU architectures, it is not guaranteed that all thread slots are actually available for use. Indeed, the resources present on the compute unit – most notable block slots, shared memory, and registers – are limited and shared between threads. Thus, if threads are particularly resource-hungry, e.g. if they require a particularly large number of registers, it may not be possible for all thread slots to be used. The corresponding metric is occupancy $q \in [0, 1]$ which describes what fraction of thread slots are usable by a given kernel. A value of $q = 1$ is optimal and indicates that all thread slots can be used. A (somewhat unlikely) value of $q = 0$ means that the kernel is not executable. For any given GPGPU kernel, the occupancy q can be easily calculated or found using a profiler. In order to facilitate the computation of N , we will introduce an intermediate value k which describes the ratio between the required number of threads and the *available* cores, incorporating the concept of occupancy. Because k is computed differently for CPUs and GPUs, we will first define k_{GPU} as in Equation 8.7:

$$k_{\text{GPU}} = \frac{c_r}{q c_s} \quad (8.7)$$

Secondly, tail effects play an important part in both the occupancy and the latency of a kernel. Tail effects occur when, towards the end of a kernel, a small *tail* of remaining work increases the latency and decreases the occupancy of that kernel. We might imagine, for example, a kernel that takes exactly 100 ms for each individual thread. On a machine with four thread slots, running this kernel with a

⁷Readers intimately familiar with CUDA programming may cast doubt on this claim, as it is well understood that performance does not linearly increase with the number of thread slots used; indeed, increasing thread slots does not increase the number of cores. Rather, using more thread slots allows the hardware to more efficiently hide latency from e.g. memory accesses. However, we must recall that the metric N simply refers to how many tasks can be ran concurrently, not how *efficiently* those tasks can be run. Therefore, we posit that the number of thread slots is, in fact, the correct metric. Any latency improvements achieved by context-switching are accounted for by the metric of latency L .

⁸In NVIDIA nomenclature, Compute Capability (CC) refers to a set of low-level performance and programmability characteristics of a device; no similar concept exists in CPU hardware.

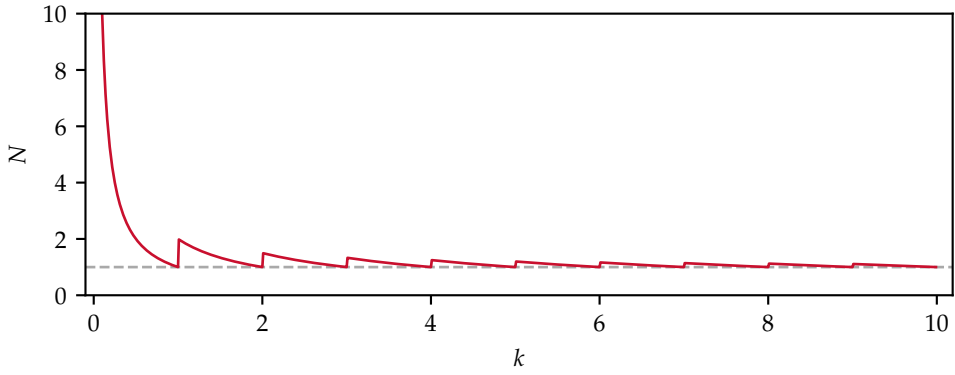


Figure 8.14: The parallelism metric N as a function of k , the ratio between the number of threads required by the application and the number of cores available on the device. Note the sawtooth pattern, as well as the tendency of N to approach 1.

single thread would incur latency of 100 ms. Running the same kernel with two, three, or four threads would incur an identical latency⁹, but executing the kernel with five threads would incur a latency of 200 ms. Indeed, four of those five threads would be able to execute in parallel, but the fifth thread would only be able to start executing after the others have finished, effectively doubling the latency: the fifth thread gives rise to a tail effect. Any measurement of kernel latency inherently includes tail effects, as latency is defined as the time between the starting and ending time of the kernel. Because L in Equation 8.6 incorporates tail effects, N must also incorporate them in order for T to be accurate. Thus, we calculate N as in Equation 8.8:

$$N = \frac{\lceil k \rceil}{k} \quad (8.8)$$

We note several corollaries of Equation 8.8. Firstly, N always has a value of at least 1 which matches the intuition that – in order to achieve maximum throughput in scenarios in which input requirements are fulfilled – it never makes sense *not* to run a kernel. Furthermore, as c_r grows larger compared to c_s , e.g. as kernels grow larger, N approaches 1 as shown in Figure 8.14; this matches the intuition that larger kernels make more efficient use of available hardware as they are less affected by tail effects. Finally, for kernels where $c_r < qc_s$, i.e. $k < 1$, the value of N can be arbitrarily high. To illustrate how Equation 8.8 can be used, we refer to an

⁹Barring any additional kernels running on the device taking up thread slots.

earlier example of a thread with 5 threads worth of work running on hardware capable of simultaneously executing 4 threads; we will assume that theoretical occupancy is 100 %. According to Equation 8.8, we obtain $k = 5/4 = 1.25$ and $N = 2/1.25 = 1.6$. If, instead, the kernel would require 9 threads worth of work, we would instead achieve $N = 3/2.25 = 4/3$; because this kernel suffers from fewer tail effects, it averages a higher hardware utilisation and thus achieves a lower value for N . Note that if c_r is a multiple of q_{cs} , it must be the case that $N = 1$ and the throughput is simply the inverse of the latency.

Compared to the GPGPU case, computing the parallelism factor N for CPU devices is simple. Indeed, such devices have no concept of occupancy, which allows k_{CPU} to be calculated trivially from the number of thread c_r and the number of cores c_d as in Equation 8.9; the computation of N remains the same as in Equation 8.8.

$$k_{\text{CPU}} = \frac{c_r}{c_d} \quad (8.9)$$

We have now established methods to estimate the throughput of kernels – running on Central Processing Units (CPUs) or on General-Purpose Graphics Processing Units (GPGPUs) – for which the throughput cannot be easily measured, as long as the latency can. With our model for the performance of a task graph as well as our model for the throughput of individual kernels in such task graphs now completed, we are ready to apply these models in practice.

8.5 Results

We have now summarised our approaches to track finding in massively parallel environments and we have proposed a methodology for evaluating these approaches in the face of effective heterogeneous scheduling methods. To round off the final chapter of this thesis, we will proceed to combine all of the aforementioned to provide a comprehensive evaluation of the performance of our approaches. To that end, we provide performance results for our algorithms and evaluate these results against the requirements posed by future colliders.

8.5.1 Kernel Benchmarks

The model discussed in Section 8.4.2 relies on the throughput of individual kernels as calculated in Section 8.4.3, which in turn relies on measurements of the latency, occupancy, and grid sizes. In this section, we employ a profiling tool to gather

Table 8.2: The versions of the TRACCC software and its dependencies used for the benchmarks described in Section 8.5.1.

Software	Version
TRACCC	0.17.0
ACTS	27.0.0
ALGEBRA-PLUGINS	0.24.0
COVFIE	0.10.0
DETRAY	0.79.0
VECMEM	1.11.0
CUDA NVCC	12.5.1
NSIGHT COMPUTE	12.5.1
G++	11.4.1

these data for the kernels in the TRACCC application. These benchmarks were run on a machine equipped with an NVIDIA RTX A5000 GPU and an AMD EPYC 7413 CPU. We evaluated version 0.17.0 of the TRACCC software, modified to remove the sorting of tracks. The code was compiled using CUDA 12.5 and gcc 11.4; the performance was evaluated using NVIDIA NSIGHT COMPUTE 12.5. A more complete version list of all the software used in these experiments is given in Table 8.2. The application was profiled while running on a series of one hundred $\mu = \langle 200 \rangle t\bar{t}$ events in the Open Data Detector (ODD) [302] simulated using GEANT4 [303]. Note that $\mu = \langle 200 \rangle$ events are a useful approximation for the data expected in the HL-LHC era. The NVIDIA RTX A5000 is a *Compute Capability* 8.6 device featuring 64 processing units with 1536 thread slots per unit for a total of 98 304 thread slots.

In total, 7453 kernels were profiled, an average of 74.5 kernels per event. Of these kernels, there were 20 unique kernels, i.e. kernels with distinct implementations. The total execution time of these kernels was 12.5 s, an average of 124.9 ms per event. Note that NSIGHT COMPUTE disables frequency boosting in order to provide the most consistent results. The aggregated results per unique kernel are shown in Table 8.3. The results for the kernels executed by the combinatorial Kálmán filter are shown in Figure 8.15 instead, as they are executed multiple times per event, and because each iteration of these kernels has significantly different performance characteristics.

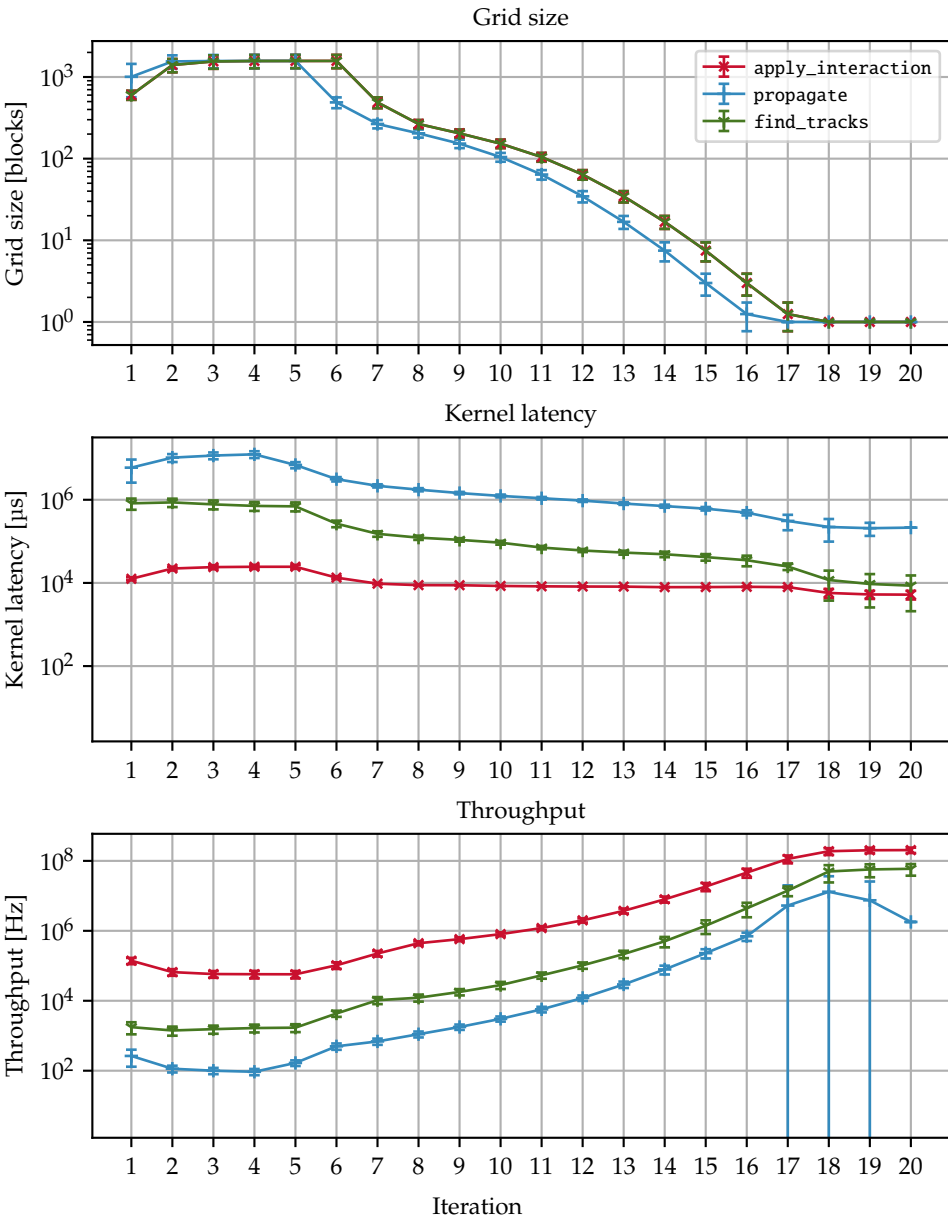


Figure 8.15: The grid size, kernel latency, and throughput for the kernels invoked by the combinatorial Kálmán filter.

8.5.2 Task Graph Performance

Using the data presented in Table 8.3 and Figure 8.15, we can finally construct a throughput model of the application as a whole, which we will present in this section. Note that we have opted to use only GPGPU-based algorithms in this model, omitting CPU-based algorithms; we consider extending the model with profiled CPU algorithms future work. The resulting flow graph consists of 162 nodes and 241 edges¹⁰ and produces a theoretical throughput of 11.12 Hz. Although this result – which is comparable to a modern, fully utilised CPU – is promising, it will need to be improved significantly to meet the requirements of the HL-LHC era. Indeed, the projected full-event tracking rate in the *Phase 2* ATLAS experiment is approximately 150 kHz¹¹; at the current achieved performance, this would require approximately 13 500 GPUs [314].

For massively parallel track reconstruction to become viable for use in the HL-LHC era, we will need to further increase the performance of the TRACCC application. To this end, our throughput model is able to guide the direction of future developments. Indeed, if our model is able to predict the throughput of our application as it stands, it can also evaluate the throughput of the application following hypothetical improvements to individual kernels. In order to demonstrate the viability of our model for this use case, we have modified each of the 20 kernels with 100 relative performance increase factors distributed logarithmically on the range 10^{-2} to 10^2 , i.e. between a *decrease* in throughput by a factor one hundred and an increase by a factor one hundred. The results of this experiment are shown in Figure 8.16. We find that these data provide a clear path forward in terms of optimisation. In particular, it is clear that the greatest performance increases can be gained by improving the performance of the `propagate` and `fit` kernels. Most of the other kernels have a much smaller, often negligible, impact on application performance.

8.6 Reproducibility and Reusability

In order to facilitate the use of our model for both descriptive purposes – i.e. the evaluation of an existing system compared to an upper bound – and prescriptive

¹⁰Not all of these nodes are useful, e.g. many transfer data from the GPU to the CPU, even though the CPU is not equipped with algorithms to process those data. Fortunately, our model works just as well with these redundant nodes and edges as without, which is why we have opted not to remove them.

¹¹In addition, there is an estimated 1 MHz rate of Region of Interest (RoI) tracking; on average, this form of tracking amounts to the reconstruction of tracks in roughly 5 % of the detector volume and takes approximately 10 % of the computing power required to reconstruct a full event [314].

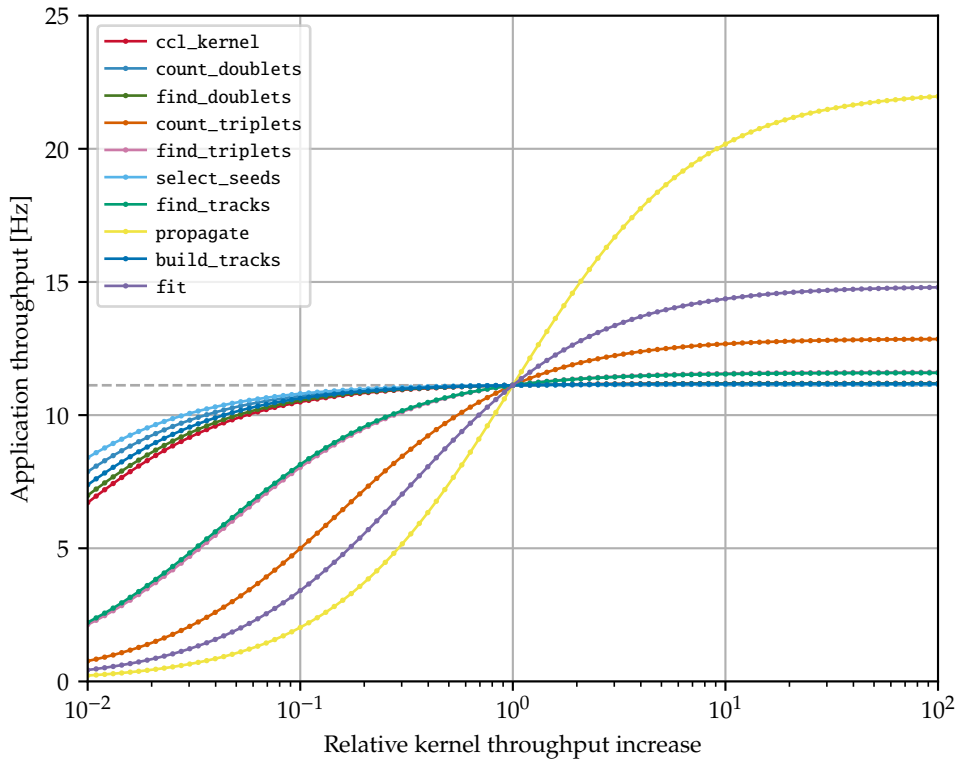


Figure 8.16: The impact on application throughput due to the increase in individual kernel throughput as described by our model. In cases where one kernel was executed multiple times, all invocations of that kernel has their throughput increased uniformly. Note the sigmoidal shapes of the curves. Only kernels that impact the performance of the overall application by at least 20 % are shown for the sake of brevity.

purposes – i.e. the evaluation of potential future changes to a system – we have implemented a Python tool that computes an upper bound on the throughput of task graphs using a simple specification format. Indeed, we require the user to specify the data types, devices, interconnects, and kernels in a human-readable YAML format and our tool will automatically construct the flow network and solve the corresponding linear program. An example of the inputs accepted by our program is given in Listing 8.1. Most real-world applications – including the application in high-energy physics relevant to us – will have more elaborate specifications, but we include only a small example for the sake of brevity. Our tool is freely available under the MIT license at <https://github.com/stephenswat/lipstick>.

The TRACCC track reconstruction pipeline which we have discussed in this chapter is implemented in C++ with a variety of heterogeneous programming models, including CUDA, SYCL, and Futhark [89]. The code is made freely available under the MPL-2.0 license at <https://github.com/acts-project/traccc>. The aforementioned repository also allows users to download track reconstruction inputs such that the code can be executed everywhere, i.e. no access to proprietary data is required. TRACCC is a collaborative effort and has been developed by numerous people.

8.7 Summary

In this chapter, we have described design and implementation of the TRACCC project, a chain of algorithms designed to perform track reconstruction on massively parallel processors. TRACCC is the result of the implementation and development of many algorithms, some of which novel and others redesigned for massively parallel execution. Furthermore, the TRACCC project ties together the models and libraries described in the other chapters of this thesis. In particular, the COVFIE library – described in Chapter 5 – was designed specifically to provide high-performance magnetic fields, and the model described in Chapter 7 was developed to help us work around the imbalanced workloads that are ubiquitous in track reconstruction. In addition to the description of TRACCC, we have described a methodology for modelling the throughput of task graph applications from measurements of their constituent application, allowing for the evaluation of the performance of such applications even if proper scheduling methods are not (yet) available. Furthermore, this methodology has allowed us to answer Research Question 4.

We believe that the TRACCC project functions as a promising proof-of-concept and as an answer to our Main Research Question, demonstrating that track reconstruction as required in the HL-LHC era is, indeed, possible on massively parallel architectures. Simultaneously, it is clear that much work remains to be done to further improve the performance of the software, as the required throughputs are higher than can be reasonably achieved with the developments hitherto described in this thesis. To this end, we have demonstrated the prescriptive power of our models, i.e. their power to inform future optimisation efforts in order to maximise their effectiveness. In the case of TRACCC, it is clear that the primary candidates for optimisation are those kernels which perform numerical propagation.

In addition to the optimisation of existing kernels, we foresee that much of the future work in the area of massively parallel track reconstruction will focus on the

development of entirely new track reconstruction strategies. For example, we have worked towards methods which find tracks not through repeated propagation of tracks but rather through the combination of existing seeds. Other novel approaches include the use of Graph Neural Networks (GNNs) to discover structure in detector data [315], similarly without the use of costly numerical integration methods. We further envision that the models presented in this chapter could be further extended with data about CPU-based track reconstruction methods – which are fully supported by the model but for which we do not currently have data – to provide a more comprehensive picture of the entire heterogeneous computing environment.

Table 8.3: The grid size, block size, occupancy, kernel latency, and throughput for the 20 unique kernels in the TRACCC application. The block size and occupancy are fixed and do not vary from invocation to invocation. The grid size and latency are given as the mean values across 100 kernel invocations. The throughput is given as the mean of the per-invocation throughput derived as described in Section 8.4.3. The statistics for the kernels invoked by the combinatorial Kálmán filter are shown in Figure 8.15.

Kernel	BS	GS	Occ. (%)	Lat. (μ s)	Thr. (Hz)
ccl_kernel	256	190.3	83.3	1052.1	1676.7
form_spacepoints	1024	104.4	66.7	76.9	16 482.4
count_grid_capacities	256	328.6	100.0	43.2	28 380.1
populate_grid	256	328.6	100.0	53.8	22 786.1
fill_prefix_sum	32	2.0	33.3	87.4	5 914 812.4
count_doublets	64	642.7	66.7	626.4	2672.0
find_doublets	64	359.8	66.7	1589.7	1858.9
count_triplets	64	111 316.0	66.7	13 347.5	81.6
reduce_triplet_counts	64	359.8	66.7	20.1	144 414.7
find_triplets	64	11 731.2	66.7	4491.9	261.0
update_triplet_weights	64	15 291.9	66.7	104.6	10 621.0
select_seeds	64	359.8	33.3	457.5	3404.8
estimate_track_params	64	593.3	66.7	64.7	27 829.3
make_barcode_sequence	64	267.4	66.7	4.5	845 489.4
apply_interaction	64	—	66.7	—	—
find_tracks	64	—	25.0	—	—
propagate	64	—	25.0	—	—
build_tracks	64	1577.4	66.7	652.6	2172.3
prune_tracks	64	492.0	66.7	254.5	8882.5
fit	64	492.0	16.7	28 139.4	44.3

9

Conclusion

*Tot ziens, tot ziens, Justine;
want welke weg ik kies, hij leidt naar hier.
Geschiedenis herhaalt zich nooit,
maar rijmt altijd een keer.*

— Erik de Jong
(Musician)

Massively parallel processor architectures have proven themselves a great boon to scientific computing by providing peak theoretical throughput numbers unmatched by traditional single- or multi-core architectures. This computational prowess is not without drawbacks, however; coordinating tens of thousands of threads is no easy task from the programmer's perspective, and the architecture of most massively parallel devices makes them prone to a variety performance pitfalls which do not – or to a lesser extent – affect other architectures. Simply put, massively parallel processor architectures excel in regular workloads with sufficient parallelism, while they struggle with irregular applications. In this thesis, we have sought to answer the question whether the reconstruction of charged particle tracks in high-energy physics can efficiently exploit the capabilities of modern massively parallel processor architectures; a difficult question no doubt, as track reconstruction is almost antithetical to the regular workloads to which massively parallel devices are most suited. In spite of this challenge, we believe that we have managed to satisfyingly advance the state of the art in massively parallel track reconstruction. In this chapter, conclude our work by reflecting on the research questions posed in Section 1.2 and by presenting our outlook on the future of our work.

9.1 Research Questions

In Section 1.2, we proposed four research questions which contribute to the main research question regarding the viability of track reconstruction on massively parallel architectures. In this section, we revisit these questions and answer them using the work presented in the preceding chapters. We will first answer the four more specific research questions, after which we will answer the main research question of this thesis.

‘What are the challenges in porting state-of-the-art reconstruction algorithms to massively parallel architectures?’ (RQ1)

Research Question 1 concerns the challenges involved in porting state-of-the-art track reconstruction algorithms to massively parallel systems. Indeed, it is said that knowing is half the battle¹: knowing which challenges prevent existing solutions from performing well on massively parallel architectures will inform the design of novel algorithms, methods, and models. We have extensively examined the aforementioned challenges in Chapter 4; this examination leads us to conclude that there are two primary challenges inhibiting massively parallel track reconstruction. The first challenge is that track reconstruction algorithms frequently access large, multi-dimensional structured grid data which need to allow high throughput in massively parallel environments. The second challenge is that the workloads in track reconstruction are highly irregular, which leads to severely degraded performance in massively parallel architectures.

‘How can structured grid data be represented in order to maximally exploit the access locality of arbitrary computations, including those in track reconstruction?’ (RQ2)

Research Question 2 follows directly from our answer to Research Question 1 and concerns the storage of structured grid data. Our goal in answering this question is to find highly performant storage strategies. In an attempt to serve the high-performance computing community as a whole, we have chosen not to restrict ourselves to applications in track reconstruction, nor do we restrict ourselves to only massively parallel architectures – although the aforementioned application and architectures remain, of course, the primary focus. To this end, we have presented a novel benchmarking strategy – published under the name COVFIE –

¹A brief investigation reveals that this common idiom is originally from the American children’s television show *G.I. Joe*; while it is perhaps unconventional to quote a fictional action hero in scientific prose, we posit that the veracity of the statement remains unaffected by its original source.

in Chapter 5 which is capable of evaluating a broad variety of storage strategies across a variety of hardware platforms under arbitrary access patterns. We achieve this lofty goal by employing compile-time composition to provide a model of representing structured grid storage methods in a way that is both flexible as well as performant. Our research does not lead us to prescribe a singular method for storing structured grids on massively parallel systems; instead of providing the metaphorical fish, we provide a method for fishing: automated benchmarking.

We expand further on Research Question 2 in Chapter 6 by examining the *layout* of structured grid data, i.e. the order in which elements are stored in memory. It is well known that the choice between e.g. the row-major and column-major can significantly impact the performance of an application. We focus specifically on a generalisation of the so-called Morton layout which gives rise to a family of array layouts so large that it cannot be feasibly benchmarked exhaustively. This led us to employ genetic algorithms. In our work on generalised Morton layouts – which we have called ALEX – we demonstrate that genetic algorithms can quickly find novel array layouts that are well suited to a specific application, and that the layouts found by our genetic algorithm can provide performance benefits in real-world applications, accelerating applications by up to a factor ten. Together with the work summarised in the previous paragraph, we have developed several novel tools that allow us to better understand how to store structured grid data in track reconstruction and beyond.

‘How can the effects of thread imbalance in SIMT workloads be modelled and how can they be mitigated?’ (RQ3)

Research Question 3 also follows from Research Question 1 and concerns the modelling and mitigation of thread imbalance in SIMT architectures, a major performance pitfall in many massively parallel applications. Due to the lockstep design of many massively parallel processors (in particular, those with GPGPU-like architectures), even minor imbalances in the per-thread work distribution can lead to significant amounts of idle time and lost performance. To better understand the impact of thread imbalance, we have developed a novel model – known as SMITE – which allows us to estimate the negative impact on performance that such imbalance will have on an application. Our model uses only statistical information about the input data of the application, which allows us to employ our model during very early stages of development, saving valuable development time. Our model also helps us to understand the impact of mitigation techniques such as load balancing, thread refinement, and thread coarsening, thereby guiding the

implementation of such methods.

‘How do the extra-functional properties of novel track reconstruction algorithms, designed to exploit massive parallelism, compare to state-of-the-art solutions?’ (RQ4)

Research Question 4 is our penultimate research question and concerns the measurement of the non-functional performance of a novel implementation of track reconstruction – designed from scratch to enable efficient execution on massively parallel environments – and the comparison of this performance to that of state-of-the-art implementations on traditional single- and multi-core architectures. To this end, we have implemented a novel track reconstruction software package known as TRACCC which we describe in detail in Chapter 8. In the same chapter, we extensively investigate the performance of the kernel in the proposed solution, and we propose a novel model for combining these results into an estimate of the peak theoretical throughput of the application. Our model can also be used to evaluate the performance of future improvements and, therefore, can be used to guide optimisation.

‘Can track reconstruction be implemented efficiently on massively parallel systems?’ (MRQ)

The answers to all of the previous research questions finally lead us to an answer to our Main Research Question, which asks whether track reconstruction can be efficiently implemented on massively parallel architectures. Considering the work towards modelling, benchmarking, and implementation that has been presented in this thesis, our answer is a cautious but hopeful *yes*. It has become clear that the irregular nature of the track reconstruction problem makes it difficult to run in parallel; not only does the problem require eight distinct steps to solve, but these steps are also computationally very different. Furthermore, the individual algorithms often exhibit one or more layers of unbounded loops, which lead to significant thread imbalance. Our work towards massively parallel has required the development of novel algorithms from scratch, and it has required us to develop novel models in order to guide our developments. Using these novel algorithms and models, we have been able to show that massively parallel track reconstruction is able to match or exceed track reconstruction on traditional homogeneous, shared-memory systems in latency, throughput, and energy efficiency.

9.2 Outlook and Future Work

Although we believe that this thesis provides a foundational understanding of the challenges in massively parallel track reconstruction as well as solutions to those challenges, it does not provide a single, conclusive solution. Both the generally applicable methods presented in this thesis as well as the application-specific solutions can be further improved. In this section, we present our outlook on how the work we have done can be expanded upon in the future.

Regarding the broader scope models and methods developed, we are excited and curious to see how they will be used in the future. We look forward to seeing these models employed in other applications, and we believe they can bring performance benefits beyond applications in high-energy physics. The COVIE benchmark presented in Chapter 5 is designed to be applicable to any domain in which multi-dimensional arrays are used. We believe that our software may be used to improve performance across a broad range of applications with relatively little effort.

The ALEX method for finding array layouts using genetic algorithm – presented in Chapter 6 – has similar potential, but we also see exciting avenues for how the methodology itself can be expanded. In particular, our current approach is limited to array layouts in which bit orders are preserved, but it need not be. Indeed, this limitation only exists because modern processor architectures cannot efficiently employ arbitrary bit orderings. As processing hardware has become increasingly flexible over recent years – in particular with the popularisation of FPGAs – we believe that the use of arbitrary array layouts may become possible. If so, our method may prove a valuable tool for finding such layouts.

We also look forward to further applications of the SMITE model which we presented in Chapter 7. Currently, our model provides a way to evaluate the performance of specific algorithms on data with specific distributions, but we believe that the model can be extended to serve a more prescriptive role. For example, a common strategy in massively parallel computing is to sort data such that thread imbalance is reduced, but it is hard to predict the efficacy of such measures. We believe that our model can serve as a foundation for more complex models which could be used to evaluate different solutions a priori to their implementation and encourage hybrid approaches such as the sorting of segments of data instead of sorting the data set as a whole.

As shown in Chapter 8, the implementation of the combinatorial Kálmán filter remains the primary performance hotspot in our software. Although the perform-

ance of this algorithm on massively parallel has been improved significantly, we do not claim that our solution is optimal. Indeed, our code suffers from high degrees of branch divergence and makes extensive use of structured grid data. We believe that much of the future work towards massively parallel track reconstruction will go towards further optimising the combinatorial Kálmán filter, which may even lead to the development of entirely novel track finding strategies.

Finally, it remains to be seen how heterogeneous kernels should be scheduled in real-world high-energy physics applications. Many high-energy physics experiments use frameworks designed for scheduling on homogeneous shared-memory systems; few are prepared for the added challenge of placement in heterogeneous, distributed-memory systems. Comprehensive scheduling solutions that are compatible with heterogeneous systems are under active research in the high-energy physics community, but it is unclear in which direction the field will choose to go. Chapter 8 of this thesis provides insight into this, as our model for task graph application throughput allows us to compare the performance of different architectures. We hope that our work in this area will help guide the design of future scheduling and placement algorithms in high-energy physics software.

9.3 Final Thoughts

On the CERN site in Meyrin, Switzerland stands the building containing the Synchrocyclotron (SC): the first particle accelerator ever built at CERN. After a remarkable 33 year career – spanning from 1957 to 1990 – the SC now serves as a museum piece, open for the general public to visit as part of a guided tour. The dangerous doses of radiation – for which the building containing the SC was equipped with concrete walls several meters thick – have made way for an environment in which we welcome tourists, and the collision targets at which highly energetic protons would have been fired have made way for a thirteen-minute video aimed at educating viewers about the history of CERN. Towards the end of that production, the audience is treated to a quote by the late Maria Fidecaro – one of the longest-serving members of CERN staff:

‘I think that curiosity and the need to do research is a part of our humanity. So there will certainly be a future.’

The field of particle physics is a beautiful example of the curiosity of humanity. Even when the practical benefits are not immediately obvious, our desire to understand the universe – from the unfathomably small to the astronomically large

– is so strong that we will move mountains² to learn all that there is to learn. It is an example of how we appreciate the intrinsic value of science: the pursuit of knowledge is valuable in and of itself, regardless of its practical applications. In modern times, the pursuit of knowledge – in virtually all fields of science – is a cooperation of human minds and computer system; humans provide the creativity that is fundamental to the advancement of science, and computers provide the almost brutish arithmetic force that is required to run complex simulations and calculations. The fact that computing has become indispensable across all fields of science has placed the field of computer science in a particularly vital role. Indeed, if mathematics is the queen of science, then perhaps computer science could – with sufficient hubris – be called the prince of science.

In this thesis, we have had the honour to apply the field of computing in order to help advance – indirectly – the field of particle physics. Throughout this thesis, our goal has always been to develop general methods which can be applied not only to computing in high-energy physics, but to high-performance computing in general. Indeed, the core goal of our work is to advance the field of computing. To this end, we have developed models, methods, and algorithms which advance – if even in a small way – our understanding of high-performance computing. We hope that these developments will allow for better, more efficient applications of high-performance computing in high-energy physics and beyond, and we hope that our work may therefore help to effect future scientific advancement. In the case of high-energy physics specifically, the consequences of this thesis are more tangible; the results presented therein will directly affect the feasibility of data processing in the High-Luminosity Large Hadron Collider, and we believe that our work will be a valuable guide in the development of software for future accelerators.

²Quite literally, in the case of the Large Hadron Collider, which runs partially under the Jura mountain range.

Acknowledgements

It is easy to think that writing a PhD thesis is solitary work; the many late nights in the office as well as all the work done on the weekends, on trains and planes, and in dusty hotel rooms while travelling would certainly suggest so. The fact of the matter, however, is we are supported in everything we do in life by our colleagues, our friends, our supervisors, and our families. This thesis is no exception to that rule, and I would now like to take a few pages to dispense of the formal academic language that so precisely describes my scientific efforts and to personally thank the people who have made my journey possible and who have thereby, in one way or another, made me who I am today.

First and foremost I would like to thank my thesis advisor Ana-Lucia Varbanescu. It is without her tireless support, feedback, and enthusiasm – not to mention her uncanny ability to answer emails at any hour of the day – that I would have been unable to even start – let alone finish – the journey towards my doctorate. I do not have the words to explain the amount of effort and patience that you have had with me. As a first-generation academic, research is an incredibly daunting field with plenty of pitfalls and idiosyncrasies. Your help in navigating paper submissions, reviewer comments, conferences, teaching and so forth have been invaluable. I could not have wished for a greater advisor and I owe you my eternal gratitude for your guidance.

I would like to thank Andreas Salzburger for his continued support throughout my doctoral work. It's one thing to be a globally recognised expert in track reconstruction, but it is another to so enthusiastically share that knowledge with the students in your group. I am perpetually impressed by the ease with which you explain complex topics and how you handle anything from departmental politics to individual pull requests. You have made we feel incredibly welcome in the EP-ADP group and your help have been truly instrumental in the writing of this thesis.

My gratitude also goes out towards Andy Pimentel, who allowed me to pursue

Acknowledgements

my thesis in the PCS group at the University of Amsterdam. Although I was rarely physically at Amsterdam Science Park, you always made me feel welcomed and like a valued part of the group, and I am very grateful for that.

Additional thanks go out to my paranymphs, Odin Helder and Rahul Balasubramanian who both have, in one way or another, been my anchors during this journey. Odin, thank you for being my long-time friend. I would like you to know that your friendship has been invaluable to me, and that your enthusiasm for table-top roleplaying games was instrumental in keeping me sane during my time in France. Rahul, I fondly remember our shakshuka cookouts and our trips to the Saint-Genis-Pouilly swimming pool. Thank you both for giving me the honour to have you as my paranymphs and for helping me in the last months of my PhD journey.

I would like to thank Jaana Heikkilä, my incredible partner who stood besides me for a significant part of this journey. It is hard to overstate how much you have taught me, and I am incredibly grateful to have you in my life. I am honoured to have you by my side not only in this academic journey but in anything we do. Who could have guessed that fate would let us cross our paths in the way that it did, and that we would be so happy together ever since. The fact that you have been through the same journey that I am now wrapping up has been invaluable and deeply comforting to me.

I would also like to thank the ATLAS team at Nikhef; in particular, I owe a great debt of gratitude to Peter Kluit, Wouter Verkerke, and Ivo van Vulpen. Peter, you have been an outstandingly patient and kind mentor, and I have learned so much from you. Wouter and Ivo, you have always been there to support me in my journey, and I appreciate that greatly. I also want to extend my gratitude to Roel Aaij from the Nikhef LHCb team, without whom I would never have ended up at Nikhef in the first place and without whom I would not have even considered doing a doctorate at CERN.

Moving to CERN during the COVID-19 pandemic was a truly difficult experience, and I would like to thank Markus Elsing and James Catmore for their support and for making me feel welcome at a new institute and in a new country. I would also like to thank Shaun Roe; I will never forget that you kind-heartedly invited me to spend Christmas with your family when I could not do so with my own. My gratitude also extends to Armin Nairz, who I also greatly appreciated as a group leader. Finally, I would like to thank Attila Krasznahorkay for being my valued office mate, even if we do not always see eye to eye.

Furthermore, I want to extend my thanks to my incredible colleagues at CERN

and Nikhef; it has been the coffee breaks and laughs I have shared with you that have made my time here worth it. In no particular order, I would like to thank Paul, Kati, Andreas, Samira, Luis, Guilherme, Joana, Joshua, Gabi, Corentin, Alexander, Florian, Jonathan, Giovanni, Julian, Luthien, Santeri, Varpu, Sioni, Robert, Rosanna, Noemi, Florine, Dante, Benjamin, Lisa, Beomki, Georgiana, Xola, Felix, Hadrien, Haider, Johannes, Max, Doğa, PF, Valentina, Dimitra, Peter, Ashley, Fabrice, Neža, Fabrice, Anton, Petar, and others. Thank you for the fun and the lemons. During my time at CERN, I also had the unique opportunity to meet and discuss with the late Hans Drevermann, for which I am very grateful.

Equally supportive were my colleagues and fellow students at the Institute for Informatics at the University of Amsterdam and elsewhere in the Netherlands. Thank you, Jelle, Marco, Lukas, Kevin, Rico, Damian, Wouter, Julius, Marius, Kyrian, Pooya, Ingrid, Hendrik, Frederick, Floris-Jan, Yixian, Matthijs, and Imke. You have helped me keep a healthy dose of Dutch *nuchterheid* and *gezelligheid* in my life.

Over the years, I have had the great privilege of travelling the world to attend conferences and symposia where I have met many talented and kind-hearted scientists. I would like to extend my thanks to Jesse, Jasper, Jerit, Griffin, Albert-Jan, and everyone one else with whom I have shared either conversations about computer science, beer, or both.

I want to thank my incredible friends from my home town of Amsterdam who have always been supportive of my work. In particular, I want to thank Gunnar, Huck, Tijn, Luc, Anusha, Diederik, Anna, Stephen (the other one), Tom, Noor, Willem, and Kaydee. You have been with me for longer than almost anyone else mentioned in this section, and I am deeply grateful to know you.

I am also grateful for the friends I met in virtual worlds with whom I have formed true bonds of friendship, and through whom the world – especially during the COVID pandemic – has always felt a little closer and more connected. In particular, my thanks go out to Michelle, Sam, George, Jesse, Ola, and Simon.

Finally, I would like to thank my parents, Mieke and Steve. You have been with me quite literally from the very beginning, and I owe so, so much of who I am today to you. Thank you for always believing in me and for inspiring me to try new things. Thanks for encouraging me to move abroad even if it wasn't easy for any of us. You have loved me, cared for me, and encouraged me more than anyone else in my life, and I will be eternally grateful for that.




In addition to the many people who have personally stood beside me, I would like to thank the developers and maintainers of the countless software packages






Acknowledgements




that have allowed me to do the research presented in this thesis, as well as to typeset the thesis itself. This research stands not only on the shoulders of giants like Alan Turing or Ada Lovelace but also on the shoulders of the countless people who develop the freely available open source software which powers so much of the world around us. Your work is often invisible and sometimes goes unappreciated, but writing this thesis would never have been possible without it. In a similar vein, I would like to thank everyone who has worked on the ACTS project with me. It has been truly delightful to work in such a cool and energetic community of like-minded individuals, and I hope that the project will go even further than it already has.


Finally, I would like to acknowledge the support of the Advanced School for Computing and Imaging (ASCI). I attended several ASCI courses throughout my PhD programme which were both very educational and a great way to get to know my fellow PhD students.

List of Publications

The following is a complete list of papers published by the author during the preparation of this thesis. Papers marked with  contribute directly to the thesis. Papers marked with  won – or were nominated for – an award. Papers marked with  received one or more reproducibility badges. Contributions by the thesis author are classified according to the CRediT taxonomy [316]. It is worth noting that this list covers publications in both computer science-focused venues as well as physics-focused venues, and these fields adhere to different standards of publication: in physics, journal publications are held in much higher regard than conference and workshop papers, with the former receiving extensive review and the latter often receiving less scrutiny and having very high acceptance rates. In computer science, on the other hand, much more value is placed on conference proceedings which are extensively reviewed and have far lower acceptance rates.

1.    Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger and Attila Krasznahorkay. ‘Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Composition’. In: *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. ICPE’23. Coimbra, Portugal: Association for Computing Machinery, 2023, pp. 55–66. ISBN: 9798400700682. DOI: 10.1145/3578244.3583723. Nominated as Candidate to the Best Paper Award. Awarded the ACM Artifacts Available and ACM Artifacts Evaluated – Reusable badges. Contributions: conceptualisation, methodology, software, validation, formal analysis, investigation, data curation, writing – original draft, visualisation, project administration.
2.   Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger and Attila Krasznahorkay. ‘Using Evolutionary Algorithms to Find Cache-Friendly Generalized Morton Layouts for Arrays’. In: *Proceedings of the 2024 ACM/SPEC International Conference on Performance Engineering*. ICPE’24. London, United Kingdom: Association for Computing

- Machinery, May 2024, pp. 83–94. doi: 10.1145/3629526.3645034. Awarded the ACM Artifacts Available, ACM Artifacts Evaluated – Functional, and ACM Results Validated – Reproduced badges. Contributions: conceptualisation, methodology, software, validation, formal analysis, investigation, data curation, writing – original draft, visualisation, project administration.
3.   Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Attila Krasznahorkay and Andy D. Pimentel. ‘Modelling Performance Loss due to Thread Imbalance in Stochastic Variable-Length SIMT Workloads’. In: *2022 30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems. MASCOTS’22*. Nice, France: IEEE, 2022, pp. 137–144. doi: 10.1109/MASCOTS56607.2022.00026. Awarded the Best Presentation Award at CompSys’22. Contributions: conceptualisation, methodology, software, validation, formal analysis, investigation, data curation, writing – original draft, visualisation, project administration.
 4.  Andreas Salzburger, Attila Krasznahorkay, Beomki Yeo, Joana Niermann and Stephen Nicholas Swatman. ‘Navigation, Field Integration and Track Parameter Transport Through Detectors Using GPUs and CPUs within the ACTS R&D Project’. In: *Proceedings of the 21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT’22)*. Oct. 2022. Contributions: software, writing – review and editing.
 5.  Beomki Yeo et al. ‘tracc – GPU Track reconstruction demonstrator for HEP’. in: *Proceedings of the 7th International Connecting The Dots Workshop (CTD’22)*. May 2022. Contributions: conceptualisation, software, writing – original draft, writing – review and editing, investigation, methodology, resources, visualisation.
 6.  Noemi Calace et al. ‘Seed Finding in the Acts Software Package: Algorithms and Optimizations’. In: *Proceedings of the 8th International Connecting The Dots Workshop (CTD’23)*. Oct. 2023. Contributions: conceptualisation, software, writing – original draft, writing – review and editing, methodology, investigation.
 7.  The ATLAS Collaboration and Stephen Nicholas Swatman. ‘Software Performance of the ATLAS Track Reconstruction for LHC Run 3’. In: *Computing and Software for Big Science* 8.1 (Apr. 2024). issn: 2510-2044. doi: 10.1007/s41781-023-00111-y. Contributions: software, investigation, writing – original draft, writing – review, visualisation.

8.  Andreas Salzburger, Joana Niermann, Attila Krasznahorkay, Stephen Nicholas Swatman, Beomki Yeo and Guilherme Metelo Rita De Almeida. 'traccc – A Close-to-Single-Source Track Reconstruction Demonstrator for CPU and GPU'. in: *Proceedings of the 26th International Conference on Computing in High Energy & Nuclear Physics (CHEP'23)*. May 2023. Contributions: conceptualisation, software, writing – original draft, writing – review and editing, investigation, methodology, resources, visualisation.
9. Stephen Nicholas Swatman, Attila Krasznahorkay and Paul Gessinger-Befurt. 'Managing Heterogeneous Device Memory Using C++17 Memory Resources'. In: *Journal of Physics: Conference Series* 2438.1 (Feb. 2023). DOI: 10.1088/1742-6596/2438/1/012050. Contributions: conceptualisation, software, formal analysis, writing – original draft.
10. Rico van Stigt, Stephen Nicholas Swatman and Ana-Lucia Varbanescu. 'Isolating GPU Architectural Features Using Parallelism-Aware Microbenchmarks'. In: *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. ICPE'22. Beijing, China: Association for Computing Machinery, 2022, pp. 77–88. ISBN: 9781450391436. DOI: 10.1145/3489525.3511673. Contributions: software, writing – original draft, writing – review and editing, supervision.
11. Ana-Lucia Varbanescu, Stephen Nicholas Swatman and Anuj Pathania. 'Performance Engineering for Graduate Students: A View from Amsterdam'. In: *Proceedings of the 2023 Workshop on Education for High Performance Computing*. EduHPC'23. Denver, Colorado, United States of America: Association for Computing Machinery, 2023. DOI: 10.1145/3624062.3624102. Contributions: methodology, validation, investigation, data curation, writing – review and editing, visualisation.
12. Uraz Odyurt, Stephen Nicholas Swatman, Ana-Lucia Varbanescu and Sascha Caron. 'Reduced Simulations for High-Energy Physics, a Middle Ground for Data-Driven Physics Research'. In: *Accepted for publication in the proceedings of the 24th International Conference on Computational Science (ICCS'24)*. Malaga, Spain, 2023, pp. 84–99. DOI: 10.1007/978-3-031-63751-3_6. Contributions: validation, investigation, data curation, writing – review and editing, visualisation, resources.
13. Sudhir Malik et al. 'Software Training in HEP'. In: *Computing and Software for Big Science* 5.1 (Oct. 2021). DOI: 10.1007/s41781-021-00069-9. Contributions: investigation.

List of Publications

14. Beomki Yeo, Heather Gray, Andreas Salzburger and Stephen Nicholas Swatman. 'The derivation of Jacobian matrices for the propagation of track parameter uncertainties in the presence of magnetic fields and detector material'. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 1068 (2024). DOI: 10.1016/j.nima.2024.169734. Contributions: software, writing – review and editing.
15. Joana Niermann et al. *detray – a library for a GPU tracking geometry*. In preparation for submission to the Journal of Instrumentation. 2024. Contributions: software, writing – original draft, writing – review and editing.

List of Acronyms

ACM Association for Computing Machinery. 104, 132, 211, 212

ACTS A Common Tracking Software. 49, 50, 97, 171, 191

AES Advanced Encryption Standard. 28

ALICE A Large Ion Collider Experiment. 50

ALU Arithmetic Logic Unit. 30, 38

AMD Advanced Micro Devices. 38, 42, 44, 91, 97, 117, 123, 125–128, 130, 131, 176, 191

API Application Programming Interface. 88

ARM Acorn RISC Machine. 32, 119

ASCI Advanced School for Computing and Imaging. 210, 216

ASIC Application-Specific Integrated Circuit. 36, 132

ATLAS A Toroidal LHC ApparatuS. 17–19, 21, 22, 24, 49–52, 55, 57, 58, 60, 69, 71, 102, 168, 169, 193, 208

AVX Advanced Vector Extensions. 32, 43, 120

BMI2 Bit Manipulation Instructions 2. 91, 117

BSM Beyond Standard Model. 11

CC Compute Capability. 151, 187, 188

CCA Connected Component Analysis. 54, 162

CCL Connected Component Labelling. 54, 55, 57, 162–166

List of Acronyms

- CERN** European Organisation for Nuclear Research. ii, iv, 4, 14, 15, 19, 22, 123, 204
- CKF** Combinatorial Kálmán Filter. 68
- CMS** Compact Muon Solenoid. 19, 49–51
- COO** Coordinate List. 55, 164–166
- CPU** Central Processing Unit. vii–ix, xi, xii, 6, 36–44, 48, 51, 52, 56, 70, 77, 78, 81–83, 87, 97–99, 103, 123, 125, 127, 135, 137, 171, 182, 187, 188, 190, 191, 193, 196
- CRediT** Contribution Roles Taxonomy. 211
- CUDA** Compute Unified Device Architecture. 37, 39, 42, 44, 45, 77, 81, 82, 87, 88, 96, 97, 103, 139, 151, 153, 161, 167, 188, 191, 195
- DAS** Distributed ASCI Supercomputer. 97, 123, 151
- DLP** Data-Level Parallelism. 31–33
- FCC** Future Circular Collider. vii, xi, 51
- FIFO** First-In First-Out. 187
- FMA** Fused Multiply-Add. 42
- FPGA** Field-Programmable Gate Array. 44, 132
- GNN** Graph Neural Network. 196
- GPGPU** General-Purpose Graphics Processing Unit. vii, ix, xi, xii, 5, 6, 24, 34, 39–41, 77, 83, 101, 136, 161, 163, 167, 173, 176, 187, 188, 190, 193, 201
- GPU** Graphics Processing Unit. vii–ix, xi, 5, 27, 35–44, 77, 78, 82, 83, 97–100, 102, 103, 136, 137, 139, 141, 151, 153, 155, 157, 160, 167, 168, 175, 186, 188, 190, 191, 193, 216
- HBM** High-Bandwidth Memory. 97, 151
- HIP** Heterogeneous Interface for Portability. 44
- HL-LHC** High-Luminosity Large Hadron Collider. vii, viii, xi, xii, 18, 52, 161, 191, 193, 195, 205
- ID** Inner Detector. 21, 51, 55, 168, 169

- IEEE** Institute of Electrical and Electronics Engineers. 38, 97
- ILP** Instruction-Level Parallelism. 29–33
- ITk** Inner Tracker. 51, 168, 169
- ITS** Independent Thread Scheduling. 153, 155
- IvI** Institute for Informatics. ii
- IVP** Initial Value Problem. 66
- LHC** Large Hadron Collider. vii, viii, xi, 6, 14–19, 48, 49, 52, 205, 215–217
- LHCb** LHC Beauty. 50, 208
- Linac4** Linear Accelerator 4. 14
- LRU** Least-Recently Used. 123
- MIMD** Multiple Instructions Multiple Data. 37, 38, 55, 57, 136–138, 141–143, 147–150, 156
- MMX** MultiMedia eXtensions. 31
- MPI** Message Passing Interface. 44
- MPL** Mozilla Public License. 104, 195
- ODD** Open Data Detector. 174, 175, 191
- OpenACC** Open Accelerators. 44, 45
- OpenCL** Open Computing Language. 45
- OpenGL** Open Graphics Library. 44
- OpenMP** Open Multi-Processing. 44, 45
- OX** Ordered Crossover. 120
- PCIe** Peripheral Component Interconnect Express. 41, 97, 151
- PCS** Parallel Computing Systems. ii
- PS** Proton Synchrotron. 15
- PSB** Proton Synchrotron Booster. 14

List of Acronyms

PTX Parallel Thread eXecution. 96, 97, 151

RISC Reduced Instruction Set Computer. 29, 215

RoI Region of Interest. 193

SC Synchrocyclotron. 204

SCT Semiconductor Tracker. 22

SIMD Single Instruction Multiple Data. 7, 31, 43, 115, 119, 120, 132, 218

SIMT Single Instruction Multiple Threads. 6, 7, 37, 38, 57, 136–144, 147–151, 153, 155–157, 159, 160, 201

SM Streaming Multiprocessor. 37, 141, 187

SMT Simultaneous Multiprocessing. 34

SPS Super Proton Synchrotron. 15

SSE Streaming SIMD eXtensions. 31, 32, 119

SVE Scalable Vector Extensions. 32, 119

TLP Task-Level Parallelism. 32, 33

YAML YAML Ain't Markup Language. 185, 194, 218

Bibliography

- [1] Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger and Attila Krasznahorkay. ‘Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Composition’. In: *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. ICPE’23. Coimbra, Portugal: Association for Computing Machinery, 2023, pp. 55–66. ISBN: 9798400700682. DOI: 10.1145/3578244.3583723.
- [2] Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger and Attila Krasznahorkay. ‘Using Evolutionary Algorithms to Find Cache-Friendly Generalized Morton Layouts for Arrays’. In: *Proceedings of the 2024 ACM/SPEC International Conference on Performance Engineering*. ICPE’24. London, United Kingdom: Association for Computing Machinery, May 2024, pp. 83–94. DOI: 10.1145/3629526.3645034.
- [3] Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Attila Krasznahorkay and Andy D. Pimentel. ‘Modelling Performance Loss due to Thread Imbalance in Stochastic Variable-Length SIMT Workloads’. In: *2022 30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. MASCOTS’22. Nice, France: IEEE, 2022, pp. 137–144. DOI: 10.1109/MASCOTS56607.2022.00026.
- [4] Bradley W. Carroll and Dale A. Ostlie. *An introduction to modern astrophysics*. Cambridge University Press, 2017.
- [5] Denis O’Brien. ‘Empedocles on the Identity of the Elements’. In: *Elenchos* 37.1-2 (2016), pp. 5–32. DOI: 10.1515/elen-2016-371-202.
- [6] Sylvia Berryman. *Ancient atomism*. 2005. URL: <https://plato.stanford.edu/Entries/atomism-ancient/>.
- [7] Andrew Pickering. *Constructing quarks: A sociological history of particle physics*. University of Chicago Press, 1999.
- [8] Gerard O’Regan. *Introduction to the history of computing: a computing history primer*. Springer, 2016.
- [9] David Schneider. ‘The Exascale Era is Upon Us: The Frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second’. In: *IEEE Spectrum* 59.1 (2022), pp. 34–35. DOI: 10.1109/MSPEC.2022.9676353.
- [10] Jan van Zuylen. ‘The microscopes of Antoni van Leeuwenhoek’. In: *Journal of microscopy* 121.3 (1981), pp. 309–328.
- [11] Max Born and Emil Wolf. *Principles of optics: electromagnetic theory of propagation, interference and diffraction of light*. Elsevier, 2013.
- [12] Ernest Rutherford. ‘The structure of the atom’. In: *Philosophical Magazine* (1914), pp. 488–498.

Bibliography

- [13] *One of the first microphotographs of the disintegration of an atomic nucleus produced in a special photographic emulsion by an artificially produced high energy particle*. CERN. 2014. URL: <https://cds.cern.ch/record/2804890>.
- [14] *A typically complex high-energy interaction produced by a neutrino*. CERN. 2014. URL: <https://cds.cern.ch/record/2805131>.
- [15] Arthur Beiser. 'Nuclear emulsion technique'. In: *Reviews of modern physics* 24.4 (1952).
- [16] Donald A. Glaser and David C. Rahm. 'Characteristics of Bubble Chambers'. In: *Phys. Rev.* 97 (2 Jan. 1955), pp. 474–479. DOI: 10.1103/PhysRev.97.474.
- [17] Sally Seidel. 'Silicon strip and pixel detectors for particle physics experiments'. In: *Physics Reports* 828 (2019). Silicon strip and pixel detectors for particle physics experiments, pp. 1–34. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2019.09.003. URL: <https://www.sciencedirect.com/science/article/pii/S0370157319302923>.
- [18] The ATLAS Collaboration. *ATLAS Software and Computing HL-LHC Roadmap*. Tech. rep. Geneva: CERN, 2022. URL: <https://cds.cern.ch/record/2802918>.
- [19] The ATLAS Collaboration and Stephen Nicholas Swatman. 'Software Performance of the ATLAS Track Reconstruction for LHC Run 3'. In: *Computing and Software for Big Science* 8.1 (Apr. 2024). ISSN: 2510-2044. DOI: 10.1007/s41781-023-00111-y.
- [20] Yngve Sneen Lindal. 'Optimizing a High Energy Physics (HEP) Toolkit on Heterogeneous Architectures'. MA thesis. Institutt for datateknikk og informasjonsvitenskap, 2011.
- [21] Krste Asanović et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec. 2006. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.
- [22] Carsten Burgard. *Example: Standard model of physics*. 2016. URL: <https://texample.net/tikz/examples/model-physics/>.
- [23] Mark Thomson. *Modern particle physics*. Cambridge University Press, 2013.
- [24] S. Abachi et al. 'Observation of the Top Quark'. In: *Phys. Rev. Lett.* 74 (14 Apr. 1995), pp. 2632–2637. DOI: 10.1103/PhysRevLett.74.2632.
- [25] K. Kodama et al. 'Observation of tau neutrino interactions'. In: *Physics Letters B* 504.3 (2001), pp. 218–224. ISSN: 0370-2693. DOI: 10.1016/S0370-2693(01)00307-0.
- [26] The CMS Collaboration. 'Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC'. In: *Physics Letters B* 716.1 (2012), pp. 30–61. ISSN: 0370-2693. DOI: 10.1016/j.physletb.2012.08.021.
- [27] The ATLAS Collaboration. 'Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC'. In: *Physics Letters B* 716.1 (2012), pp. 1–29. ISSN: 0370-2693. DOI: 10.1016/j.physletb.2012.08.020.
- [28] Neta A. Bahcall, Lori M. Lubin and Victoria Dorman. 'Where Is the Dark Matter?' In: *The Astrophysical Journal* 447.2 (July 1995). DOI: 10.1086/309577.
- [29] Albert Einstein. 'Does the inertia of a body depend upon its energy-content'. In: *Annalen der physik* 18.13 (1905), pp. 639–641.
- [30] Michael Benedikt and Frank Zimmermann. 'Towards future circular colliders'. In: *Journal of the Korean Physical Society* 69.6 (Sept. 2016), pp. 893–902. ISSN: 1976-8524. DOI: 10.3938/jkps.69.893.
- [31] Ronald D. Ruth. 'The next linear collider'. In: *Frontiers of Particle Beams: Intensity Limitations*. Ed. by M. Dienes, M. Month and S. Turner. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 562–591. ISBN: 978-3-540-46797-7.

- [32] Jeffrey Forshaw and Gavin Smith. *Dynamics and relativity*. John Wiley & Sons, 2014.
- [33] H. Schmickler. *Colliders*. 2024. arXiv: 2011.01638 [physics.acc-ph].
- [34] Lyndon Evans. ‘The Large Hadron Collider’. In: *New Journal of Physics* 9.9 (Sept. 2007). doi: 10.1088/1367-2630/9/9/335.
- [35] L Arnaudon et al. *Linac4 Technical Design Report*. Tech. rep. Geneva: CERN, 2006. URL: <https://cds.cern.ch/record/1004186>.
- [36] Paul Gessinger-Befurt. ‘Development and improvement of track reconstruction software and search for disappearing tracks with the ATLAS experiment’. PhD thesis. Johannes Gutenberg-Universität Mainz, 2021.
- [37] K H Reich. *The CERN proton synchrotron booster*. Tech. rep. CERN, 1969. URL: <https://cds.cern.ch/record/349912>.
- [38] J. P. Blewett. ‘The Proton Synchrotron’. In: *Reports on Progress in Physics* 19.1 (Jan. 1956). doi: 10.1088/0034-4885/19/1/302.
- [39] J.B. Adams and E.J.N. Wilson. ‘Design studies for a large proton synchrotron and its laboratory’. In: *Nuclear Instruments and Methods* 87.2 (1970), pp. 157–179. issn: 0029-554X. doi: 10.1016/0029-554X(70)90199-0.
- [40] Michael Benedikt, Paul Collier, V Mertens, John Poole and Karlheinz Schindl. *LHC Design Report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2004. doi: 10.5170/CERN-2004-003-V-3. URL: <https://cds.cern.ch/record/823808>.
- [41] The ATLAS Collaboration. *ATLAS public beam spot results*. 2018. URL: <https://twiki.cern.ch/twiki/bin/view/AtlasPublic/BeamSpotPublicResults>.
- [42] The ATLAS Collaboration. *Interactions per Crossing 2011-2023*. 2023. URL: <https://twiki.cern.ch/twiki/bin/view/AtlasPublic/LuminosityPublicResultsRun3>.
- [43] The ATLAS Collaboration. ‘Measurement of the inelastic proton–proton cross-section at $\sqrt{s}=7$ TeV with the ATLAS detector’. In: *Nature Communications* 2.1 (Sept. 2011). issn: 2041-1723. doi: 10.1038/ncomms1472.
- [44] The ATLAS Collaboration. ‘Combined measurement of the total and differential cross sections in the $H \rightarrow \gamma\gamma$ and the $H \rightarrow ZZ^* \rightarrow 4\ell$ decay channels at $\sqrt{s}=13$ TeV with the ATLAS detector’. In: *Physics Letters B* 786 (2018), pp. 114–133. issn: 0370-2693. doi: 10.1016/j.physletb.2018.09.019.
- [45] Andreas Salzburger. ‘Optimisation of the ATLAS Track Reconstruction Software for Run-2’. In: *Journal of Physics: Conference Series* 664.7 (Dec. 2015). doi: 10.1088/1742-6596/664/7/072042.
- [46] *Computer generated image of the whole ATLAS detector*. CERN. 2008. URL: <https://cds.cern.ch/record/1095924>.
- [47] The ATLAS Collaboration. *ATLAS Fact Sheet*. 2011. URL: <https://atlas.cern/Resources/Fact-sheets>.
- [48] Marzena Lapka, David Barney and Achintya Rao. *CMS Factsheet*. 2016. URL: <https://cds.cern.ch/record/2204857>.
- [49] Christian Lippmann. ‘Detector Physics of Resistive Plate Chambers’. PhD thesis. Johann Wolfgang Goethe-Universität Frankfurt am Main, 2003.
- [50] Ziheng Chen, Antonio Di Pilato, Felice Pantaleo and Marco Rovere. ‘GPU-based Clustering Algorithm for the CMS High Granularity Calorimeter’. In: *EPJ Web Conf.* 245 (2020). doi: 10.1051/epjconf/202024505005.

Bibliography

- [51] Nuno Fernandes and The ATLAS Collaboration. 'GPU acceleration of the ATLAS calorimeter clustering algorithm'. In: *Journal of Physics: Conference Series* 2438.1 (Feb. 2023). DOI: 10.1088/1742-6596/2438/1/012044.
- [52] A. Aloisio et al. 'The trigger chambers of the ATLAS muon spectrometer: production and tests'. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 535.1 (2004). Proceedings of the 10th International Vienna Conference on Instrumentation, pp. 265–271. ISSN: 0168-9002. DOI: 10.1016/j.nima.2004.07.130.
- [53] Valerie Halyo, Patrick LeGresley, Paul Lujan, Vadim Karpusenko and Alexey Vladimirov. 'First evaluation of the CPU, GPGPU and MIC architectures for real time particle tracking based on Hough transform at the LHC'. In: *Journal of Instrumentation* 9.04 (Apr. 2014). DOI: 10.1088/1748-0221/9/04/P04005.
- [54] The ATLAS Collaboration. 'Muon reconstruction performance of the ATLAS detector in proton–proton collision data at $\sqrt{s}=13$ TeV'. In: *The European Physical Journal C* 76.5 (May 2016). ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-016-4120-y.
- [55] *ATLAS event at 900 GeV – 6 May 2015 – Run 264034 Evt 11475271*. CERN. 2015. URL: <https://cds.cern.ch/record/2014666>.
- [56] The ATLAS Collaboration. 'ATLAS pixel detector electronics and sensors'. In: *Journal of Instrumentation* 3.07 (July 2008). DOI: 10.1088/1748-0221/3/07/P07007.
- [57] A. La Rosa. 'ATLAS IBL Pixel Upgrade'. In: *Nuclear Physics B - Proceedings Supplements* 215.1 (2011). Proceedings of the 12th Topical Seminar on Innovative Particle and Radiation Detectors (IPRD10), pp. 147–150. ISSN: 0920-5632. DOI: 10.1016/j.nucphysbps.2011.03.161.
- [58] Marilena Bandieramonte, Riccardo Maria Bianchi, Joseph Boudreau, Andrea Dell'Acqua and Vakhtang Tsulaia. 'The GeoModel tool suite for detector description'. In: *EPJ Web Conf.* 251 (2021). DOI: 10.1051/epjconf/202125103007.
- [59] M. Frank, F. Gaede, C. Grefe and P. Mato. 'DD4hep: A Detector Description Toolkit for High Energy Physics Experiments'. In: *Journal of Physics: Conference Series* 513.2 (June 2014). DOI: 10.1088/1742-6596/513/2/022010.
- [60] Andreas Salzburger, Joana Niermann, Beomki Yeo and Attila Krasznahorkay. 'Detray: a compile time polymorphic tracking geometry description'. In: *Journal of Physics: Conference Series* 2438.1 (Feb. 2023). DOI: 10.1088/1742-6596/2438/1/012026.
- [61] Gordon Earle Moore. 'Cramming more components onto integrated circuits'. In: *Electronics* 35.8 (Apr. 1965).
- [62] M. V. Wilkes. 'The CMOS end-point and related topics in computing'. English. In: *Computing & Control Engineering Journal* 7 (2 Apr. 1996), pp. 101–106. ISSN: 0956-3385. DOI: 10.1049/cce_19960207.
- [63] Karl Rupp. *42 Years of Microprocessor Trend Data*. Feb. 2018. URL: <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>.
- [64] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. 2020. URL: <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>.
- [65] Paul A. Packan. 'Pushing the Limits'. In: *Science* 285.5436 (1999), pp. 2079–2081. DOI: 10.1126/science.285.5436.2079.
- [66] James E. Thornton. 'The CDC 6600 Project'. In: *Annals of the History of Computing* 2.4 (1980), pp. 338–348. DOI: 10.1109/MAHC.1980.10044.

- [67] K. Skadron, P.S. Ahuja, M. Martonosi and D.W. Clark. 'Branch prediction, instruction-window size, and cache size: performance trade-offs and simulation techniques'. In: *IEEE Transactions on Computers* 48.11 (1999), pp. 1260–1281. doi: 10.1109/12.811115.
- [68] Daniele Folegnani and Antonio González. 'Energy-effective issue logic'. In: *Proceedings of the 28th Annual International Symposium on Computer Architecture*. ISCA'01. Göteborg, Sweden: Association for Computing Machinery, 2001, pp. 230–239. ISBN: 0769511627. doi: 10.1145/379240.379266.
- [69] Weiji Guo. 'Efficient Constant-Time Implementation of SM4 with Intel GFNI instruction set extension and Arm NEON coprocessor'. In: *Cryptology ePrint Archive* (2022).
- [70] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous and A.R. LeBlanc. 'Design of ion-implanted MOSFET's with very small physical dimensions'. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. doi: 10.1109/JSSC.1974.1050511.
- [71] L. Johnsson and G. Netzer. 'The impact of Moore's Law and loss of Dennard scaling: Are DSP SoCs an energy efficient alternative to x86 SoCs?' In: *Journal of Physics: Conference Series* 762.1 (Oct. 2016). doi: 10.1088/1742-6596/762/1/012022.
- [72] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [73] A. Hartstein and Thomas R. Puzak. 'The optimum pipeline depth for a microprocessor'. In: *SIGARCH Comput. Archit. News* 30.2 (May 2002), pp. 7–13. ISSN: 0163-5964. doi: 10.1145/545214.545217.
- [74] Johannes Hofmann, Dietmar Fey, Jan Eitzinger, Georg Hager and Gerhard Wellein. 'Analysis of Intel's Haswell Microarchitecture Using the ECM Model and Microbenchmarks'. In: *Architecture of Computing Systems – ARCS 2016*. Ed. by Frank Hannig, João M. P. Cardoso, Thilo Pionteck, Dietmar Fey, Wolfgang Schröder-Preikschat and Jürgen Teich. Cham: Springer International Publishing, 2016, pp. 210–222. ISBN: 978-3-319-30695-7.
- [75] J.E. Smith and G.S. Sohi. 'The microarchitecture of superscalar processors'. In: *Proceedings of the IEEE* 83.12 (1995), pp. 1609–1624. doi: 10.1109/5.476078.
- [76] José González and Antonio González. 'Speculative execution via address prediction and data prefetching'. In: *Proceedings of the 11th International Conference on Supercomputing*. ICS'97. Vienna, Austria: Association for Computing Machinery, 1997, pp. 196–203. ISBN: 0897919025. doi: 10.1145/263580.263631.
- [77] Michael J. Flynn. 'Very high-speed computing systems'. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. doi: 10.1109/PROC.1966.5273.
- [78] D.A. Draper et al. 'An X86 microprocessor with multimedia extensions'. In: *1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers*. 1997, pp. 172–173. doi: 10.1109/ISSCC.1997.585321.
- [79] Keith Diefendorff. 'Katmai enhances MMX'. In: *Microprocessor Report* 12.13 (1998), pp. 1–4.
- [80] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman and Donald E. Porter. 'x86-64 Instruction United States of Americage among C/C++ Applications'. In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. SYSTOR'19. Haifa, Israel: Association for Computing Machinery, 2019, pp. 68–79. ISBN: 9781450367493. doi: 10.1145/3319647.3325833.
- [81] Nigel Stephens et al. 'The ARM Scalable Vector Extension'. In: *IEEE Micro* 37.2 (2017), pp. 26–39. doi: 10.1109/MM.2017.35.

Bibliography

- [82] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn and Wolfgang E Nagel. ‘Auto-vectorization techniques for modern SIMD architectures’. In: *Proceedings of the 16th Workshop on Compilers for Parallel Computing, Padova, Italy (January 2012)*. 2012.
- [83] Jules Penuchot, Joel Falcou and Amal Khabou. ‘Modern Generative Programming for Optimizing Small Matrix-Vector Multiplication’. In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. 2018, pp. 508–514. doi: 10.1109/HPCS.2018.00086.
- [84] Pierre Estérie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté and Brigitte Rozoy. ‘Boost.SIMD: Generic programming for portable SIMDization’. In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 431–432.
- [85] Joel Falcou and Jocelyn Sérot. ‘EVE, an Object Oriented SIMD Library’. In: *Computational Science - ICCS 2004*. Ed. by Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 314–321. isbn: 978-3-540-24688-6.
- [86] Charles R. Harris et al. ‘Array programming with NumPy’. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. issn: 1476-4687. doi: 10.1038/s41586-020-2649-2.
- [87] Adin D. Falkoff and Kenneth E. Iverson. ‘The evolution of APL’. In: *SIGPLAN Notes* 13.8 (Aug. 1978), pp. 47–57. issn: 0362-1340. doi: 10.1145/960118.808372.
- [88] Jeff Bezanson, Stefan Karpinski, Viral B. Shah and Alan Edelman. *Julia: A Fast Dynamic Language for Technical Computing*. 2012. arXiv: 1209.5145 [cs.PL].
- [89] Troels Henriksen. ‘Design and implementation of the Futhark programming language’. PhD thesis. University of Copenhagen, Faculty of Science, 2017.
- [90] Gene M. Amdahl. ‘Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities’. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference. AFIPS’67*. Atlantic City, New Jersey, United States of America: Association for Computing Machinery, 1967, pp. 483–485. isbn: 9781450378956. doi: 10.1145/1465482.1465560.
- [91] John L. Gustafson. ‘Reevaluating Amdahl’s law’. In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. issn: 0001-0782. doi: 10.1145/42411.42415.
- [92] Jeff Parsons. *The PCjs Project*. 2020. URL: <https://www.pcjs.org/>.
- [93] Yuhsuke Koyama. ‘Arcade Games (1): From Elemechaelemecha to Video Games’. In: *History of the Japanese Video Game Industry*. Singapore: Springer Nature Singapore, 2023, pp. 15–25. isbn: 978-981-99-1342-8. doi: 10.1007/978-981-99-1342-8_2.
- [94] Peter Shirley, Michael Ashikhmin and Steve Marschner. *Fundamentals of computer graphics*. AK Peters/CRC Press, 2009.
- [95] Advanced Micro Devices, Inc. *AMD EPYC 9754*. Advanced Micro Devices, Inc. 2023. URL: <https://www.amd.com/en/products/cpu/amd-epyc-9754>.
- [96] Jack Choquette. ‘NVIDIA Hopper H100 GPU: Scaling Performance’. In: *IEEE Micro* 43.03 (May 2023), pp. 9–17. issn: 1937-4143. doi: 10.1109/MM.2023.3256796.
- [97] *CUDA C++ Programming Guide*. NVIDIA Corporation. 2021. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [98] NVIDIA Corporation. *NVIDIA Ampere GA102 GPU Architecture*. 2020. URL: <https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf>.
- [99] Advanced Micro Devices, Inc. *AMD CDNA 2 Architecture*. 2022. URL: <https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf>.

- [100] William J. Dally, Stephen W. Keckler and David B. Kirk. 'Evolution of the Graphics Processing Unit (GPU)'. In: *IEEE Micro* 41.6 (2021), pp. 42–51. doi: 10.1109/MM.2021.3113475.
- [101] David B. Kirk and Wen-Mei W. Hwu. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.
- [102] Erik Lindholm, John Nickolls, Stuart Oberman and John Montrym. 'NVIDIA Tesla: A Unified Graphics and Computing Architecture'. In: *IEEE Micro* 28.2 (2008), pp. 39–55. doi: 10.1109/MM.2008.31.
- [103] Tianyi David Han and Tarek S. Abdelrahman. 'Reducing branch divergence in GPU programs'. In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. GPGPU-4*. Newport Beach, California, United States of America: Association for Computing Machinery, 2011. ISBN: 9781450305693. doi: 10.1145/1964179.1964184.
- [104] Sparsh Mittal and Jeffrey S. Vetter. 'A Survey of CPU-GPU Heterogeneous Computing Techniques'. In: *ACM Computing Surveys* 47.4 (July 2015), pp. 1–35. issn: 0360-0300. doi: 10.1145/2788396.
- [105] Anne C. Elster and Tor A. Haugdahl. 'NVIDIA Hopper GPU and Grace CPU Highlights'. In: *Computing in Science & Engineering* 24.2 (2022), pp. 95–100. doi: 10.1109/MCSE.2022.3163817.
- [106] Patsy A. McLaughlin and Rafael Lemaitre. 'Carcinization in the Anomura - fact or fiction? I. Evidence from adult morphology'. In: *Contributions to Zoology* 67.2 (1997), pp. 79–123. doi: 10.1163/18759866-06702001.
- [107] Victor W. Lee et al. 'Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU'. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture. ISCA'10*. Saint-Malo, France: Association for Computing Machinery, 2010, pp. 451–460. ISBN: 9781450300537. doi: 10.1145/1815961.1816021.
- [108] Nikolay Kondratyuk, Vsevolod Nikolskiy, Daniil Pavlov and Vladimir Stegailov. 'GPU-accelerated molecular dynamics: State-of-art software performance and porting from Nvidia CUDA to AMD HIP'. In: *The International Journal of High Performance Computing Applications* 35.4 (2021), pp. 312–324. doi: 10.1177/10943420211008288.
- [109] Graham Sellers and John Kessenich. *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016.
- [110] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.
- [111] Rob Farber. *Parallel programming with OpenACC*. Newnes, 2016.
- [112] Ronan Keryell, Ruyman Reyes and Lee Howes. 'Khronos SYCL for OpenCL: a tutorial'. In: *Proceedings of the 3rd International Workshop on OpenCL. IWOCCL'15*. Palo Alto, California, United States of America: Association for Computing Machinery, 2015. ISBN: 9781450334846. doi: 10.1145/2791321.2791345.
- [113] Christian R. Trott et al. 'Kokkos 3: Programming Model Extensions for the Exascale Era'. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. doi: 10.1109/TPDS.2021.3097283.
- [114] Erik Zenker et al. 'Alpaka – An Abstraction Library for Parallel Kernel Acceleration'. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 631–640. doi: 10.1109/IPDPSW.2016.50.
- [115] Bradford L. Chamberlain, Steve Deitz, Mary Beth Hribar and Wayne Wong. 'Chapel'. In: *Programming Models for Parallel Computing* (2015), pp. 129–159.
- [116] Aaftab Munshi, Benedict Gaster, Timothy G Mattson and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.

Bibliography

- [117] Jianbin Fang, Ana Lucia Varbanescu and Henk Sips. ‘A Comprehensive Performance Comparison of CUDA and OpenCL’. In: *2011 International Conference on Parallel Processing*. 2011, pp. 216–225. doi: 10.1109/ICPP.2011.45.
- [118] FUJIFILM Corporation. *FUJICHROME Velvia 50 Professional [RVP50]*. Apr. 2008.
- [119] The CMS Collaboration. *The CMSSW Documentation Suite*. Dec. 2006. URL: <https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBook>.
- [120] The ATLAS Collaboration. *ATLAS: Computing Technical Proposal*. Tech. rep. Geneva: CERN, Dec. 1996. URL: <https://cds.cern.ch/record/322322>.
- [121] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron and A. Mockus. ‘Does code decay? Assessing the evidence from change management data’. In: *IEEE Transactions on Software Engineering* 27.1 (2001), pp. 1–12. doi: 10.1109/32.895984.
- [122] Xiaocong Ai et al. ‘A Common Tracking Software Project’. In: *Computing and Software for Big Science* 6.1 (Apr. 2022). issn: 2510-2044. doi: 10.1007/s41781-021-00078-8.
- [123] Joseph D. Osborn et al. ‘Implementation of ACTS into sPHENIX Track Reconstruction’. In: *Computing and Software for Big Science* 5.1 (Oct. 2021). issn: 2510-2044. doi: 10.1007/s41781-021-00068-w.
- [124] Henso Abreu et al. ‘First Direct Observation of Collider Neutrinos with FASER at the LHC’. In: *Phys. Rev. Lett.* 131 (3 July 2023). doi: 10.1103/PhysRevLett.131.031801.
- [125] Lawrence, David. ‘EIC Software Overview’. In: *EPJ Web of Conferences* 295 (2024). doi: 10.1051/epjconf/202429503011.
- [126] Xiaocong Ai, Georgiana Mania, Heather M. Gray, Michael Kuhn and Nicholas Styles. ‘A GPU-Based Kalman Filter for Track Fitting’. In: *Computing and Software for Big Science* 5.1 (Oct. 2021). issn: 2510-2044. doi: 10.1007/s41781-021-00065-z.
- [127] Cerati, Giuseppe et al. ‘Parallelized Kalman-Filter-Based Reconstruction of Particle Tracks on Many-Core Processors and GPUs’. In: *EPJ Web of Conferences* 150 (2017). doi: 10.1051/epjconf/201715000006.
- [128] Sergey Gorbunov et al. ‘ALICE HLT High Speed Tracking on GPU’. In: *IEEE Transactions on Nuclear Science* 58.4 (2011), pp. 1845–1851. doi: 10.1109/TNS.2011.2157702.
- [129] David Rohr, Sergey Gorbunov, Volker Lindenstruth and for the ALICE Collaboration. ‘GPU-accelerated track reconstruction in the ALICE High Level Trigger’. In: *Journal of Physics: Conference Series* 898.3 (Oct. 2017). doi: 10.1088/1742-6596/898/3/032030.
- [130] Daniel Hugo Cámpora Pérez, Niko Neufeld and Agustin Riscos Nuñez. ‘A Fast Local Algorithm for Track Reconstruction on Parallel Architectures’. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2019, pp. 698–707. doi: 10.1109/IPDPSW.2019.00118.
- [131] R. Aaij et al. ‘Allen: A High-Level Trigger on GPUs for LHCb’. In: *Computing and Software for Big Science* 4.1 (Apr. 2020). issn: 2510-2044. doi: 10.1007/s41781-020-00039-7.
- [132] A. Bocci, V. Innocente, M. Kortelainen, F. Pantaleo and M. Rovere. ‘Heterogeneous Reconstruction of Tracks and Primary Vertices With the CMS Pixel Tracker’. In: *Frontiers in Big Data* 3 (2020). issn: 2624-909X. doi: 10.3389/fdata.2020.601728.
- [133] S. Amerio et al. ‘Applications of GPUs to online track reconstruction in HEP experiments’. In: *2012 IEEE Nuclear Science Symposium and Medical Imaging Conference Record (NSS/MIC)*. 2012, pp. 1806–1811. doi: 10.1109/NSSMIC.2012.6551422.

- [134] Michael Benedikt, Alain Blondel, Patrick Janot, Michelangelo Mangano and Frank Zimmermann. ‘Future Circular Colliders succeeding the LHC’. In: *Nature Physics* 16.4 (Apr. 2020), pp. 402–407. ISSN: 1745-2481. DOI: 10.1038/s41567-020-0856-2.
- [135] Lingxin Meng. *ATLAS ITk Pixel Detector Overview*. 2021. arXiv: 2105.10367 [physics.ins-det].
- [136] K Mathieson et al. ‘Charge sharing in silicon pixel detectors’. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 487.1 (2002). 3rd International Workshop on Radiation Imaging Detectors, pp. 113–122. ISSN: 0168-9002. DOI: 10.1016/S0168-9002(02)00954-3.
- [137] Costantino Grana, Daniele Borghesani and Rita Cucchiara. ‘Connected Component Labeling Techniques on Modern Architectures’. In: *Image Analysis and Processing – ICIAP 2009*. Ed. by Pasquale Foggia, Carlo Sansone and Mario Vento. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 816–824. ISBN: 978-3-642-04146-4. DOI: 10.1007/978-3-642-04146-4_87.
- [138] Kiehn, Moritz et al. ‘The TrackML high-energy physics tracking challenge on Kaggle’. In: *EPJ Web of Conferences* 214 (2019). DOI: 10.1051/epjconf/201921406037.
- [139] Eduardo Ros. ‘ATLAS Inner Detector’. In: *Nuclear Physics B - Proceedings Supplements* 120 (2003). Proceedings of the 8th International Conference on B-Physics at Hadron Machines, pp. 235–238. ISSN: 0920-5632. DOI: 10.1016/S0920-5632(03)01908-X.
- [140] Daniel Langr and Pavel Tvrdík. ‘Evaluation Criteria for Sparse Matrix Storage Formats’. In: *IEEE Transactions on Parallel and Distributed Systems* 27.2 (2016), pp. 428–440. DOI: 10.1109/TPDS.2015.2401575.
- [141] Arthur Hennequin, Ben Couturier, Vladimir V. Gligorov and Lionel Lacassagne. ‘SparseCCL: Connected Components Labeling and Analysis for sparse images’. In: *2019 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2019, pp. 65–70. DOI: 10.1109/DASIP48288.2019.9049184.
- [142] Arthur Hennequin. ‘Performance optimization for the LHCb experiment’. PhD thesis. Sorbonne Université, Jan. 2022.
- [143] The ATLAS Collaboration. ‘A neural network clustering algorithm for the ATLAS silicon pixel detector’. In: *Journal of Instrumentation* 9.09 (Sept. 2014). DOI: 10.1088/1748-0221/9/09/P09009.
- [144] Corentin Allaire. ‘A High-Granularity Timing Detector in ATLAS: Performance at the HL-LHC’. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 924 (2019), pp. 355–359. ISSN: 0168-9002. DOI: 10.1016/j.nima.2018.05.028.
- [145] Rocky Bala Garg, Corentin Allaire, Andreas Salzburger, Hadrien Grasland, Lauren Tompkins and Elyssa Hofgard. ‘Potentiality of automatic parameter tuning suite available in ACTS track reconstruction software framework’. In: *EPJ Web of Conferences* 295 (2024). DOI: 10.1051/epjconf/202429503031.
- [146] Veikko Karimäki. ‘Effective circle fitting for particle trajectories’. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 305.1 (1991), pp. 187–191. ISSN: 0168-9002. DOI: 10.1016/0168-9002(91)90533-V.
- [147] Rudolf Emil Kálmán. ‘A New Approach to Linear Filtering and Prediction Problems’. In: *Journal of Basic Engineering* 82.1 (Mar. 1960), pp. 35–45. ISSN: 0021-9223. DOI: 10.1115/1.3662552.

Bibliography

- [148] Donald Eugene. Groom and Spencer R. Klein. 'Passage of particles through matter'. In: *The European Physical Journal C - Particles and Fields* 15.1 (Mar. 2000), pp. 163–173. issn: 1434-6052. doi: 10.1007/BF02683419.
- [149] Carl Runge. 'Über die numerische Auflösung von Differentialgleichungen'. In: *Mathematische Annalen* 46.2 (1895), pp. 167–178. doi: 10.1007/BF01446807.
- [150] Erwin Fehlberg. 'Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittweiten-Kontrolle und ihre Anwendung auf Wärmeleitungsprobleme'. In: *Computing* 6.1 (Mar. 1970), pp. 61–71. issn: 1436-5057. doi: 10.1007/BF02241732.
- [151] Fabian Klimpel. 'Generalisation of track extrapolation, reconstruction and parametrised simulation for the ACTS project in the context of experiment-comprehensive, future tracking challenges using the example of the ATLAS detector'. PhD thesis. Technische Universität München, 2021. URL: <https://mediatum.ub.tum.de/1616441>.
- [152] D. Alspach and H. Sorenson. 'Nonlinear Bayesian estimation using Gaussian sum approximations'. In: *IEEE Transactions on Automatic Control* 17.4 (1972), pp. 439–448. doi: 10.1109/TAC.1972.1100034.
- [153] Samuel David Jones and on behalf of the ATLAS Collaboration. 'The ATLAS Electron and Photon Trigger'. In: *Journal of Physics: Conference Series* 1162.1 (Jan. 2019), p. 012037. doi: 10.1088/1742-6596/1162/1/012037.
- [154] Thijs Gerrit Cornelissen. 'Track Fitting in the ATLAS Experiment'. PhD thesis. University of Amsterdam, 2006. URL: <https://cds.cern.ch/record/1005181>.
- [155] Stephen Nicholas Swatman. 'Performance Engineering in High Energy Physics Software: The ATLAS Offline χ^2 Track Fitter'. Master's Thesis. University of Amsterdam, 2019. URL: <https://cds.cern.ch/record/2875725>.
- [156] Corentin Allaire, Françoise Bouvet, Hadrien Grasland and David Rousseau. 'Ranking-based neural network for ambiguity resolution in ACTS'. In: *EPJ Web of Conferences* 295 (2024). doi: 10.1051/epjconf/202429503022.
- [157] Moritz Lehmann, Mathias J. Krause, Giorgio Amati, Marcello Sega, Jens Harting and Stephan Gekle. 'Accuracy and performance of the lattice Boltzmann method with 64-bit, 32-bit, and customized 16-bit number formats'. In: *Phys. Rev. E* 106 (1 July 2022). doi: 10.1103/PhysRevE.106.015308.
- [158] Steffen Raach, David Schlipf, Florian Haizmann and Po Wen Cheng. 'Three Dimensional Dynamic Model Based Wind Field Reconstruction from Lidar Data'. In: *Journal of Physics: Conference Series* 524 (June 2014). doi: 10.1088/1742-6596/524/1/012005.
- [159] Ki Myung Brian Lee, Chanyeol Yoo, Ben Hollings, Stuart Anstee, Shoudong Huang and Robert Fitch. 'Online Estimation of Ocean Current from Sparse GPS Data for Underwater Vehicles'. In: *2019 International Conference on Robotics and Automation (ICRA)*. Montreal, QC, Canada: IEEE, 2019, pp. 3443–3449. doi: 10.1109/ICRA.2019.8794308.
- [160] Fabian Klimpel. 'Track propagation for different detector and magnetic field setups in Acts'. In: *Journal of Physics: Conference Series* 1525.1 (Apr. 2020). doi: 10.1088/1742-6596/1525/1/012080.
- [161] Martin Glinz. 'On Non-Functional Requirements'. In: *15th IEEE International Requirements Engineering Conference (RE 2007)*. Delhi, India: IEEE, Oct. 2007, pp. 21–26. doi: 10.1109/RE.2007.45.

- [162] Rainer Buchty, Vincent Heuveline, Wolfgang Karl and Jan-Philipp Weiss. 'A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators'. In: *Concurrency and Computation: Practice and Experience* 24.7 (2012), pp. 663–675. doi: 10.1002/cpe.1904.
- [163] Antonio Galbis and Manuel Maestre. 'Vectors and Vector Fields'. In: *Vector Analysis Versus Vector Calculus*. Boston, MA: Springer US, 2012, pp. 1–17. ISBN: 978-1-4614-2200-6. doi: 10.1007/978-1-4614-2200-6_1.
- [164] Jeyarajan Thiyagalingam, Olav Beckmann and Paul H. J. Kelly. 'Is Morton layout competitive for large two-dimensional arrays yet?' In: *Concurrency and Computation: Practice and Experience* 18.11 (2006), pp. 1509–1539. doi: 10.1002/cpe.1018.
- [165] Anthony E. Nocentino and Philip J. Rhodes. 'Optimizing Memory Access on GPUs Using Morton Order Indexing'. In: *Proceedings of the 48th Annual Southeast Regional Conference*. ACM SE'10. Oxford, Mississippi: Association for Computing Machinery, 2010. ISBN: 9781450300643. doi: 10.1145/1900008.1900035.
- [166] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala and Mithuna Thottethodi. 'Recursive array layouts and fast matrix multiplication'. In: *IEEE Transactions on Parallel and Distributed Systems* 13.11 (2002), pp. 1105–1123. doi: 10.1109/TPDS.2002.1058095.
- [167] Sunita Sarawagi and Michael Stonebraker. 'Efficient organization of large multidimensional arrays'. In: *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*. Houston, Texas, United States of America: IEEE, 1994, pp. 328–336. doi: 10.1109/ICDE.1994.283048.
- [168] H. Carter Edwards and Daniel Sunderland. 'Kokkos Array Performance-Portable Manycore Programming Model'. In: *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM'12. New Orleans, Louisiana: Association for Computing Machinery, 2012, pp. 1–10. ISBN: 9781450312110. doi: 10.1145/2141702.2141703.
- [169] Martin Hofmann. 'Syntax and semantics of dependent types'. In: *Extensional Constructs in Intensional Type Theory*. London: Springer London, 1997, pp. 13–54. ISBN: 978-1-4471-0963-1. doi: 10.1007/978-1-4471-0963-1_2.
- [170] Donald E. Knuth. 'Two Notes on Notation'. In: *The American Mathematical Monthly* 99.5 (1992), pp. 403–422. doi: 10.1080/00029890.1992.11995869.
- [171] Peter Dimov. *Boost.Mp11: A C++11 metaprogramming library*. The Boost Organization. 2017. URL: <https://www.boost.org/doc/libs/master/libs/mp11/doc/html/mp11.html>.
- [172] Google LLC. *BENCHMARK: a microbenchmark support library*. 2016. URL: <https://github.com/google/benchmark>.
- [173] John Charles Butcher. 'Numerical Differential Equation Methods'. In: *Numerical Methods for Ordinary Differential Equations*. Chichester, West Sussex, England: John Wiley & Sons, Ltd, 2016. Chap. 2, pp. 55–142. ISBN: 9781119121534. doi: 10.1002/9781119121534.ch2.
- [174] Bartosz Milewski. *Category theory for programmers*. Aug. 2019.
- [175] Takayuki Okimura, Teruyoshi Sasayama, Norio Takahashi and Soichiro Ikuno. 'Parallelization of Finite Element Analysis of Nonlinear Magnetic Fields Using GPU'. In: *IEEE Transactions on Magnetics* 49.5 (2013), pp. 1557–1560. doi: 10.1109/TMAG.2013.2244062.
- [176] Yash Ukidave et al. 'NUPAR: A Benchmark Suite for Modern GPU Architectures'. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE'15. Austin, Texas, United States of America: Association for Computing Machinery, 2015, pp. 253–264. ISBN: 9781450332484. doi: 10.1145/2668930.2688046.

Bibliography

- [177] Mahmut Taylan Kandemir, Alok Choudhary, J. Ramanujam, N. Shenoy and Prithviraj Banerjee. 'Enhancing spatial locality via data layout optimizations'. In: *Euro-Par'98 Parallel Processing*. Ed. by David Pritchard and Jeff Reeve. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 422–434. ISBN: 9783540499206. DOI: 10.1007/BFb0057885.
- [178] Jonathan K. Lawder. 'The application of space-filling curves to the storage and retrieval of multi-dimensional data'. PhD thesis. University of London, 2000.
- [179] K. Patrick Lorton and David S. Wise. 'Analyzing Block Locality in Morton-Order and Morton-Hybrid Matrices'. In: *SIGARCH Comput. Archit. News* 35.4 (Sept. 2007), pp. 6–12. ISSN: 0163-5964. DOI: 10.1145/1327312.1327315.
- [180] Daniel Kusswurm. 'Advanced Vector Extensions (AVX)'. In: *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*. Berkeley, California, United States of America: Apress, 2014, pp. 327–349. ISBN: 978-1-4842-0064-3. DOI: 10.1007/978-1-4842-0064-3_12.
- [181] Agner Fog et al. 'Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs'. In: *Copenhagen University College of Engineering* 93 (2011).
- [182] David Hilbert. 'Ueber die stetige Abbildung einer Linie auf ein Flächenstück'. In: *Mathematische Annalen* 38.3 (1891), pp. 459–460. DOI: 10.1007/BF01199431.
- [183] Jonathan K. Lawder and Peter J. H. King. 'Using Space-Filling Curves for Multi-dimensional Indexing'. In: *Advances in Databases*. Ed. by Brian Lings and Keith Jeffery. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 20–35. ISBN: 978-3-540-45033-7. DOI: 10.1007/3-540-45033-5_3.
- [184] Bongki Moon, H. V. Jagadish, Christos Faloutsos and Joel Saltz. 'Analysis of the clustering properties of the Hilbert space-filling curve'. In: *IEEE Transactions on Knowledge and Data Engineering* 13.1 (2001), pp. 124–141. DOI: 10.1109/69.908985.
- [185] Patrick Franco, Giap Nguyen, Remy Mullot and Jean-Marc Ogier. 'Alternative patterns of the multidimensional Hilbert curve'. In: *Multimedia Tools and Applications* 77.7 (2018), pp. 8419–8440. DOI: 10.1007/s11042-017-4744-4.
- [186] Herman Haverkort. *An inventory of three-dimensional Hilbert space-filling curves*. 2011. DOI: 10.48550/ARXIV.1109.2323.
- [187] Mahmut Taylan Kandemir, J. Ramanujam and Alok Choudhary. 'Improving cache locality by a combination of loop and data transformations'. In: *IEEE Transactions on Computers* 48.2 (1999), pp. 159–167. DOI: 10.1109/12.752657.
- [188] Paul Bourke. 'Interpolation methods'. Dec. 1999.
- [189] Karel Driesen and Urs Hölzle. 'The Direct Cost of Virtual Function Calls in C++'. In: *SIGPLAN Not.* 31.10 (Oct. 1996), pp. 306–323. ISSN: 0362-1340. DOI: 10.1145/236338.236369.
- [190] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis and M. Siegle. 'Compositional performance modelling with the TIPTool'. In: *Performance Evaluation* 39.1 (2000), pp. 5–35. ISSN: 0166-5316. DOI: 10.1016/S0166-5316(99)00056-5.
- [191] Murali Sitaraman. 'Compositional performance reasoning'. In: *Procs. Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*. Citeseer. 2001, pp. 3–10.
- [192] Advanced Micro Devices, Inc. *AMD EPYC 7402P*. Advanced Micro Devices, Inc. 2019. URL: <https://www.amd.com/en/products/cpu/amd-epyc-7402p>.
- [193] David Suggs, Mahesh Subramony and Dan Bouvier. 'The AMD "Zen 2" Processor'. In: *IEEE Micro* 40.2 (2020), pp. 45–52. DOI: 10.1109/MM.2020.2974217.

- [194] Henri Bal et al. 'A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term'. In: *Computer* 49.05 (May 2016), pp. 54–63. ISSN: 1558-0814. DOI: 10.1109/MC.2016.127.
- [195] Intel Corporation. *Intel Xeon Processor E5-2630 v3*. Intel Corporation. 2014. URL: <https://ark.intel.com/content/www/us/en/ark/products/83356/intel-xeon-processor-e52630-v3-20m-cache-2-40-ghz.html>.
- [196] Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger and Attila Krasznahorkay. *Benchmarking Software for "Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Composition"*. Jan. 2023. DOI: 10.5281/zenodo.7019829.
- [197] Stephen Nicholas Swatman. *COVFIE: a Compositional Vector Field Library*. 2022. URL: <https://github.com/acts-project/covfie>.
- [198] Krste Asanović et al. 'A View of the Parallel Computing Landscape'. In: *Communications of the ACM* 52.10 (Oct. 2009), pp. 56–67. ISSN: 0001-0782. DOI: 10.1145/1562764.1562783.
- [199] Neungsoo Park, Bo Hong and Viktor K. Prasanna. 'Tiling, block data layout, and memory hierarchy performance'. In: *IEEE Transactions on Parallel and Distributed Systems* 14.7 (2003), pp. 640–654. DOI: 10.1109/TPDS.2003.1214317.
- [200] Guy Macdonald Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. Tech. rep. International Business Machines Company, 1966.
- [201] John Henry Holland. 'Genetic algorithms'. In: *Scientific American* 267.1 (1992), pp. 66–73.
- [202] Tamara G. Kolda and Brett W. Bader. 'Tensor Decompositions and Applications'. In: *SIAM Review* 51.3 (2009), pp. 455–500. DOI: 10.1137/07070111X.
- [203] Markus Kowarschik and Christian Weiß. 'An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms'. In: *Algorithms for Memory Hierarchies: Advanced Lectures*. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg, 2003, pp. 213–232. ISBN: 978-3-540-36574-7. DOI: 10.1007/3-540-36574-5_10.
- [204] Jonathan Weinberg, Michael O. McCracken, Erich Strohmaier and Allan Snaveley. 'Quantifying Locality In The Memory Access Patterns of HPC Applications'. In: *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. Seattle, Washington, United States of America: IEEE, 2005, pp. 50–50. DOI: 10.1109/SC.2005.59.
- [205] Byunghyun Jang, Dana Schaa, Perhaad Mistry and David Kaeli. 'Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures'. In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (2011), pp. 105–118. DOI: 10.1109/TPDS.2010.107.
- [206] Shuai Che, Jeremy W. Sheaffer and Kevin Skadron. 'Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems'. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC'11. Seattle, Washington: Association for Computing Machinery, 2011. ISBN: 9781450307710. DOI: 10.1145/2063384.2063401.
- [207] Ibrahim Al-Kharusi and David W. Walker. 'Locality properties of 3D data orderings with application to parallel molecular dynamics simulations'. In: *The International Journal of High Performance Computing Applications* 33.5 (2019), pp. 998–1018. DOI: 10.1177/1094342019846282.
- [208] Martin Perdacher, Claudia Plant and Christian Böhm. 'Improved Data Locality Using Morton-order Curve on the Example of LU Decomposition'. In: *2020 IEEE International Conference on Big Data (Big Data)*. Atlanta, Georgia, United States of America: IEEE Computer Society, 2020, pp. 351–360. DOI: 10.1109/BigData50022.2020.9378385.

Bibliography

- [209] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala and Mithuna Thottethodi. 'Recursive Array Layouts and Fast Parallel Matrix Multiplication'. In: *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA'99. Saint Malo, France: Association for Computing Machinery, 1999, pp. 222–231. ISBN: 1581131240. DOI: 10.1145/305619.305645.
- [210] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra and Mithuna Thottethodi. 'Nonlinear Array Layouts for Hierarchical Memory Systems'. In: *Proceedings of the 13th International Conference on Supercomputing*. ICS'99. Rhodes, Greece: Association for Computing Machinery, 1999, pp. 444–453. ISBN: 158113164X. DOI: 10.1145/305138.305231.
- [211] Filip Pawłowski, Bora Uçar and Albert-Jan Nicholas Yzelman. 'A multi-dimensional Morton-ordered block storage for mode-oblivious tensor computations'. In: *Journal of Computational Science* 33 (2019), pp. 34–44. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2019.02.007.
- [212] John Mellor-Crummey, David Whalley and Ken Kennedy. 'Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings'. In: *International Journal of Parallel Programming* 29.3 (June 2001), pp. 217–247. ISSN: 1573-7640. DOI: 10.1023/A:1011119519789.
- [213] Steven T. Gabriel and David S. Wise. 'The Opie Compiler from Row-Major Source to Morton-Ordered Matrices'. In: *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*. WMPI'04. Munich, Germany: Association for Computing Machinery, 2004, pp. 136–144. ISBN: 159593040X. DOI: 10.1145/1054943.1054962.
- [214] David W. Walker and Anthony Skjellum. *The Impact of Space-Filling Curves on Data Movement in Parallel Systems*. 2023. DOI: 10.48550/arXiv.2307.07828. arXiv: 2307.07828 [cs.DC].
- [215] Daryl Deford and Ananth Kalyanaraman. 'Empirical Analysis of Space-Filling Curves for Scientific Computing Applications'. In: *2013 42nd International Conference on Parallel Processing*. Lyon, France: IEEE, 2013, pp. 170–179. DOI: 10.1109/ICPP.2013.26.
- [216] Michael Bader. *Space-filling curves: an introduction with applications in scientific computing*. Vol. 9. Berlin, Heidelberg, Germany: Springer Science & Business Media, 2012.
- [217] Michael Armbrust et al. 'Delta lake: high-performance ACID table storage over cloud object stores'. In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 3411–3424. ISSN: 2150-8097. DOI: 10.14778/3415478.3415560.
- [218] OEIS Foundation Inc. *Moser–de Bruijn sequence, entry A000695 in The On-Line Encyclopedia of Integer Sequences*. 2023. URL: <https://oeis.org/A000695>.
- [219] Peter Gottschling, David S. Wise and Michael D. Adams. 'Representation-Transparent Matrix Algorithms with Scalable Performance'. In: *Proceedings of the 21st Annual International Conference on Supercomputing*. ICS'07. Seattle, Washington: Association for Computing Machinery, 2007, pp. 116–125. ISBN: 9781595937681. DOI: 10.1145/1274971.1274989.
- [220] Peter Gottschling, David S. Wise and Adwait Joshi. 'Generic support of algorithmic and structural recursion for scientific computing'. In: *International Journal of Parallel, Emergent and Distributed Systems* 24.6 (2009), pp. 479–503. DOI: 10.1080/17445760902758560.
- [221] David W. Walker. 'Morton ordering of 2D arrays for efficient access to hierarchical memory'. In: *The International Journal of High Performance Computing Applications* 32.1 (2018), pp. 189–203. DOI: 10.1177/1094342017725568.

- [222] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. Cambridge, Massachusetts, United States of America: MIT Press, 1992.
- [223] G. Pavai and T. V. Geetha. 'A Survey on Crossover Operators'. In: *ACM Comput. Surv.* 49.4 (Dec. 2016). issn: 0360-0300. doi: 10.1145/3009966.
- [224] M. Srinivas and Lalit M. Patnaik. 'Genetic algorithms: a survey'. In: *Computer* 27.6 (1994), pp. 17–26. doi: 10.1109/2.294849.
- [225] S.N. Sivanandam and S.N. Deepa. 'Genetic Algorithms'. In: *Introduction to Genetic Algorithms*. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg, 2008, pp. 15–37. isbn: 978-3-540-73190-0. doi: 10.1007/978-3-540-73190-0_2.
- [226] Lothar Terfloth and Johann Gasteiger. 'Neural networks and genetic algorithms in drug design'. In: *Drug Discovery Today* 6 (2001), pp. 102–108. issn: 1359-6446. doi: 10.1016/S1359-6446(01)00173-8.
- [227] Andrew Gartland-Jones and Peter Copley. 'The Suitability of Genetic Algorithms for Musical Composition'. In: *Contemporary Music Review* 22.3 (2003), pp. 43–55. doi: 10.1080/0749446032000150870.
- [228] Andy D. Pimentel. 'Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration'. In: *IEEE Design & Test* 34.1 (2017), pp. 77–90. doi: 10.1109/MDAT.2016.2626445.
- [229] Dolly Sapra and Andy D. Pimentel. 'Constrained Evolutionary Piecemeal Training to Design Convolutional Neural Networks'. In: *Trends in Artificial Intelligence Theory and Applications: Artificial Intelligence Practices*. Cham, Switzerland: Springer International Publishing, 2020, pp. 709–721. isbn: 978-3-030-55789-8.
- [230] Edward J. Anderson and Michael C. Ferris. 'Genetic Algorithms for Combinatorial Optimization: The Assemble Line Balancing Problem'. In: *ORSA Journal on Computing* 6.2 (1994), pp. 161–173. doi: 10.1287/ijoc.6.2.161.
- [231] Heinz Mühlenbein. 'Parallel genetic algorithms, population genetics and combinatorial optimization'. In: *Parallelism, Learning, Evolution*. Wildbad Kreuth, Germany: Springer Berlin Heidelberg, 1991, pp. 398–406. isbn: 978-3-540-46663-5.
- [232] Brian Hegerty, Chih-Cheng Hung and Kristen Kasprak. 'A comparative study on differential evolution and genetic algorithms for some combinatorial problems'. In: *Proceedings of the 8th Mexican International Conference on Artificial Intelligence*. Vol. 9. Guanajuato, Mexico: Springer Verlag, 2009.
- [233] José Fernando Gonçalves and Mauricio G. C. Resende. 'Biased random-key genetic algorithms for combinatorial optimization'. In: *Journal of Heuristics* 17.5 (Oct. 2011), pp. 487–525. issn: 1572-9397. doi: 10.1007/s10732-010-9143-1.
- [234] Richard A. Brualdi. *Introductory combinatorics*. 5th ed. London, United Kingdom: Pearson Education, 1977. isbn: 978-0136020400.
- [235] João Nuno Ferreira Alves, Luís Manuel Silveira Russo and Alexandre Francisco. 'Cache-Oblivious Hilbert Curve-Based Blocking Scheme for Matrix Transposition'. In: *ACM Trans. Math. Softw.* 48.4 (Dec. 2022). issn: 0098-3500. doi: 10.1145/3555353.
- [236] Albert-Jan Nicholas Yzelman and Rob Hendrik Bisseling. 'A Cache-Oblivious Sparse Matrix-Vector Multiplication Scheme Based on the Hilbert Curve'. In: *Progress in Industrial Mathematics at ECMI 2010*. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg, 2012, pp. 627–633. isbn: 978-3-642-25100-9.

Bibliography

- [237] Andreas Abel and Jan Reineke. ‘uops.info: Characterizing Latency, Throughput, and Port United States of Americage of Instructions on Intel Microarchitectures’. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS’19. Providence, Rhode Island, United States of America: Association for Computing Machinery, 2019, pp. 673–686. ISBN: 9781450362405. DOI: 10.1145/3297858.3304062.
- [238] Per Hammarlund et al. ‘Haswell: The Fourth-Generation Intel Core Processor’. In: *IEEE Micro* 34.2 (2014), pp. 6–20. DOI: 10.1109/MM.2014.10.
- [239] Mark Evers, Leslie Barnes and Mike Clark. ‘The AMD Next-Generation “Zen 3” Core’. In: *IEEE Micro* 42.3 (2022), pp. 7–12. DOI: 10.1109/MM.2022.3152788.
- [240] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager and Gerhard Wellein. ‘Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures’. In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. Dallas, Texas, United States of America: IEEE, 2018, pp. 121–131. DOI: 10.1109/PMBS.2018.8641578.
- [241] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. June 2023.
- [242] Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran. ‘Cache-oblivious algorithms’. In: *40th Annual Symposium on Foundations of Computer Science*. FOCS’99. New York, New York, United States of America: IEEE, 1999, pp. 285–297. DOI: 10.1109/SFFCS.1999.814600.
- [243] Adam Slowik and Halina Kwasnicka. ‘Evolutionary algorithms and their applications to engineering problems’. In: *Neural Computing and Applications* 32.16 (Aug. 2020), pp. 12363–12379. ISSN: 1433-3058. DOI: 10.1007/s00521-020-04832-8.
- [244] Lawrence Davis. ‘Applying Adaptive Algorithms to Epistatic Domains’. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI’85. Los Angeles, California, United States of America: Morgan Kaufmann Publishers Inc., 1985, pp. 162–164. ISBN: 0934613028.
- [245] Agoston E. Eiben and Jim E. Smith. ‘Representation, Mutation, and Recombination’. In: *Introduction to Evolutionary Computing*. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg, 2015, pp. 49–78. ISBN: 978-3-662-44874-8. DOI: 10.1007/978-3-662-44874-8_4.
- [246] Brad L. Miller and David E. Goldberg. ‘Genetic algorithms, tournament selection, and the effects of noise’. In: *Complex systems* 9.3 (1995), pp. 193–212.
- [247] Julian Hammer, Jan Eitzinger, Georg Hager and Gerhard Wellein. ‘Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels’. In: *Tools for High Performance Computing 2016*. Ed. by Christoph Niethammer, José Gracia, Tobias Hilbrich, Andreas Knüpfer, Michael M. Resch and Wolfgang E. Nagel. Cham: Springer International Publishing, 2017, pp. 1–22. ISBN: 978-3-319-56702-0. DOI: 10.1007/978-3-319-56702-0_1.
- [248] Wenzel Jakob, Jason Rhinelander and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. 2017. URL: <https://github.com/pybind/pybind11>.
- [249] Ryutaro Himeno. *The Riken Himeno CFD Benchmark*. 2001. URL: <https://i.riken.jp/en/supercom/documents/himenobmt/>.
- [250] Intel Corporation. *Intel Xeon Processor E5-2660 v3*. Intel Corporation. 2014. URL: <https://ark.intel.com/content/www/us/en/ark/products/81706/intel-xeon-processor-e52660-v3-25m-cache-2-60-ghz.html>.

- [251] Advanced Micro Devices, Inc. *AMD EPYC 7413*. Advanced Micro Devices, Inc. 2014. URL: <https://www.amd.com/en/products/cpu/amd-epyc-7413>.
- [252] Pepe Vila, Pierre Ganty, Marco Guarnieri and Boris Köpf. 'CacheQuery: Learning Replacement Policies from Hardware Caches'. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'20. London, UK: Association for Computing Machinery, 2020, pp. 519–532. ISBN: 9781450376136. DOI: 10.1145/3385412.3386008.
- [253] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely and Joel Emer. 'Set-Dueling-Controlled Adaptive Insertion for High-Performance Caching'. In: *IEEE Micro* 28.1 (2008), pp. 91–98. DOI: 10.1109/MM.2008.14.
- [254] Advanced Micro Devices, Inc. *Software optimization guide for the AMD family 19h processors*. Nov. 2020.
- [255] Vali Codreanu, Joerg Hertzer, Cristian Morales, Jorge Rodriguez, Ole Widar Saastad and Martin Stachon. *Best practice guide Haswell / Broadwell*. Ed. by Volker Weinberg. Jan. 2017.
- [256] Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy D. Pimentel, Andreas Salzburger and Attila Krasznahorkay. *Artifact for "Using Evolutionary Algorithms to Find Cache-Friendly Generalized Morton Layouts for Arrays"*. Feb. 2024. DOI: 10.5281/zenodo.10567243.
- [257] Stephen Nicholas Swatman. *ALEX: an Array Layout Evolution eXperiment*. 2023. URL: <https://github.com/stephenswat/alex>.
- [258] John D. Owens et al. 'A Survey of General-Purpose Computation on Graphics Hardware'. In: *Computer Graphics Forum* 26.1 (Mar. 2007), pp. 80–113. DOI: 10.1111/j.1467-8659.2007.01012.x.
- [259] Piotr Bialas and Adam Strzelecki. 'Benchmarking the cost of thread divergence in CUDA'. In: *International Conference on Parallel Processing and Applied Mathematics*. Krakow, Poland: Springer International Publishing, 2016, pp. 570–579. DOI: 10.1007/978-3-319-32149-3_53.
- [260] Ping Xiang, Yi Yang and Huiyang Zhou. 'Warp-level divergence in GPUs: Characterization, impact, and mitigation'. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Orlando, Florida, United States of America: IEEE, 2014, pp. 284–295. DOI: 10.1109/HPCA.2014.6835939.
- [261] John Runwei Cheng and Mitsuo Gen. 'Accelerating genetic algorithms with GPU computing: A selective overview'. In: *Computers & Industrial Engineering* 128 (2019), pp. 514–525. ISSN: 0360-8352. DOI: 10.1016/j.cie.2018.12.067.
- [262] Lena Oden and Jörg Keller. 'Improving Cryptanalytic Applications with Stochastic Runtimes on GPUs'. In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Portland, Oregon, United States of America: IEEE, 2021, pp. 459–468. DOI: 10.1109/IPDPSW52791.2021.00077.
- [263] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi and Kunle Olukotun. 'Accelerating CUDA Graph Algorithms at Maximum Warp'. In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPoPP'11. San Antonio, Texas, United States of America: Association for Computing Machinery, 2011, pp. 267–276. ISBN: 9781450301190. DOI: 10.1145/1941553.1941590.
- [264] Lawrence Murray. 'GPU Acceleration of Runge–Kutta Integrators'. In: *IEEE Transactions on Parallel and Distributed Systems* 23.1 (2012), pp. 94–101. DOI: 10.1109/TPDS.2011.61.
- [265] Mark Harris and Kyrlyo Perelygin. *Cooperative Groups: Flexible CUDA Thread Programming*. 2017. URL: <https://developer.nvidia.com/blog/cooperative-groups/>.

Bibliography

- [266] The Khronos SYCL Working Group. *SYCL 2020 Specification (revision 4)*. Khronos Group. 2021. URL: <https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf>.
- [267] Steffen Frey, Guido Reina and Thomas Ertl. 'SIMT microscheduling: Reducing thread stalling in divergent iterative algorithms'. In: *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. Munich, Germany: IEEE, 2012, pp. 399–406. DOI: 10.1109/PDP.2012.62.
- [268] Yossi Shiloach and Uzi Vishkin. 'An $O(n^2 \log n)$ parallel max-flow algorithm'. In: *Journal of Algorithms* 3.2 (1982), pp. 128–146. ISSN: 0196-6774. DOI: 10.1016/0196-6774(82)90013-X.
- [269] H. N. Nagaraja. 'Order Statistics from Discrete Distributions'. In: *Statistics* 23.3 (1992), pp. 189–216. DOI: 10.1080/02331889208802365.
- [270] Pier Luigi Novi Inverardi and Aldo Tagliani. 'Discrete distributions from moment generating function'. In: *Applied Mathematics and Computation* 182.1 (2006), pp. 200–209. DOI: 10.1016/j.amc.2006.03.048.
- [271] Makoto Matsumoto and Takuji Nishimura. 'Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator'. In: *ACM Trans. Model. Comput. Simul.* 8.1 (Jan. 1998), pp. 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995.
- [272] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture*. 2017. URL: <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>.
- [273] Jack Choquette, Olivier Giroux and Denis Foley. 'Volta: Performance and Programmability'. In: *IEEE Micro* 38.2 (2018), pp. 42–52. DOI: 10.1109/MM.2018.022071134.
- [274] Hartwig Anzt and Jack Dongarra. 'A Jaccard Weights Kernel Leveraging Independent Thread Scheduling on GPUs'. In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Lyon, France: IEEE, 2018, pp. 229–232. DOI: 10.1109/CAHPC.2018.8645946.
- [275] John R. Dormand and Peter J. Prince. 'A family of embedded Runge–Kutta formulae'. In: *Journal of Computational and Applied Mathematics* 6.1 (1980), pp. 19–26. ISSN: 0377-0427. DOI: 10.1016/0771-050X(80)90013-3.
- [276] Andreas Salzburger et al. *A Common Tracking Software Project*. European Organization for Nuclear Research, July 2021. DOI: 10.5281/zenodo.5141419.
- [277] Souley Madougou, Ana Varbanescu, Cees de Laat and Rob van Nieuwpoort. 'The landscape of GPGPU performance modeling tools'. In: *Parallel Computing* 56 (2016), pp. 18–33. ISSN: 0167-8191. DOI: 10.1016/j.parco.2016.04.002.
- [278] Andrew Kerr, Eric Anger, Gilbert Hendry and Sudhakar Yalamanchili. 'Eiger: A framework for the automated synthesis of statistical performance models'. In: *2012 19th International Conference on High Performance Computing*. Pune, India: IEEE, 2012, pp. 1–6. DOI: 10.1109/HiPC.2012.6507525.
- [279] Wenhao Jia, Kelly A. Shaw and Margaret Martonosi. 'Stargazer: Automated regression-based GPU design space exploration'. In: *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. New Brunswick, New Jersey, United States of America: IEEE, 2012, pp. 2–13. DOI: 10.1109/ISPASS.2012.6189201.
- [280] Ying Zhang, Yue Hu, Bin Li and Lu Peng. 'Performance and power analysis of ATI GPU: A statistical approach'. In: *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*. Dalian, China: IEEE, 2011, pp. 149–158. DOI: 10.1109/NAS.2011.51.

- [281] Farzad Khorasani, Rajiv Gupta and Laxmi N. Bhuyan. 'Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection'. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: Association for Computing Machinery, 2015, pp. 204–215. ISBN: 9781450340342. DOI: 10.1145/2830772.2830796.
- [282] Hasyim Gautama and Arjan J. C. van Gemund. 'Low-cost static performance prediction of parallel stochastic task compositions'. In: *IEEE Transactions on Parallel and Distributed Systems* 17.1 (2006), pp. 78–91. DOI: 10.1109/TPDS.2006.13.
- [283] Hasyim Gautama and Arjan J. C. van Gemund. 'Static Performance Prediction of Data-Dependent Programs'. In: *Proceedings of the 2nd International Workshop on Software and Performance*. WOSP'00. Ottawa, Ontario, Canada: Association for Computing Machinery, 2000, pp. 216–226. ISBN: 158113195X. DOI: 10.1145/350391.350435.
- [284] Alberto Magni, Christophe Dubach and Michael O'Boyle. 'Automatic optimization of thread-coarsening for graphics processors'. In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. Edmonton, Alberta, Canada: Association for Computing Machinery, 2014, pp. 455–466. DOI: 10.1145/2628071.2628087.
- [285] Alberto Magni, Christophe Dubach and Michael O'Boyle. 'A large-scale cross-architecture evaluation of thread-coarsening'. In: *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Denver, Colorado, United States of America: Association for Computing Machinery, 2013, pp. 1–11. DOI: 10.1145/2503210.2503268.
- [286] Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Attila Krasznahorkay and Andy D. Pimentel. *Artifact for "Statistically Modelling the Overhead of Thread Imbalance in Stochastic Variable-Length SMT Workloads"*. Apr. 2024. DOI: 10.5281/zenodo.10931331.
- [287] Stephen Nicholas Swatman. *SMITE: a Statistical Model for Imbalanced Thread Execution*. 2022. URL: <https://github.com/stephenswat/smite>.
- [288] The Acts Project. *tracc*. 2021. URL: <https://github.com/acts-project/tracc>.
- [289] Andreas Salzburger, Attila Krasznahorkay, Beomki Yeo, Joana Niermann and Stephen Nicholas Swatman. 'Navigation, Field Integration and Track Parameter Transport Through Detectors Using GPUs and CPUs within the ACTS R&D Project'. In: *Proceedings of the 21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT'22)*. Oct. 2022.
- [290] Beomki Yeo et al. 'tracc – GPU Track reconstruction demonstrator for HEP'. In: *Proceedings of the 7th International Connecting The Dots Workshop (CTD'22)*. May 2022.
- [291] Noemi Calace et al. 'Seed Finding in the Acts Software Package: Algorithms and Optimizations'. In: *Proceedings of the 8th International Connecting The Dots Workshop (CTD'23)*. Oct. 2023.
- [292] Andreas Salzburger, Joana Niermann, Attila Krasznahorkay, Stephen Nicholas Swatman, Beomki Yeo and Guilherme Metelo Rita De Almeida. 'tracc – A Close-to-Single-Source Track Reconstruction Demonstrator for CPU and GPU'. In: *Proceedings of the 26th International Conference on Computing in High Energy & Nuclear Physics (CHEP'23)*. May 2023.
- [293] Yongzhe Zhang, Ariful Azad and Zhenjiang Hu. 'FastSV: A Distributed-Memory Connected Component Algorithm with Fast Convergence'. In: *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing (PP)*. 2020, pp. 46–57. DOI: 10.1137/1.9781611976137.5.
- [294] Yossi Shiloach and Uzi Vishkin. 'An $O(\log n)$ parallel connectivity algorithm'. In: *Journal of Algorithms* 3.1 (1982), pp. 57–67. ISSN: 0196-6774. DOI: 10.1016/0196-6774(82)90008-6.

Bibliography

- [295] Laura Gonella and The ATLAS ITk Collaboration. 'The ATLAS ITk detector system for the Phase-II LHC upgrade'. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 1045 (2023). ISSN: 0168-9002. DOI: 10.1016/j.nima.2022.167597.
- [296] M Karagounis. *Development of the ATLAS FE-I4 pixel readout IC for b-layer Upgrade and Super-LHC*. Tech. rep. CERN, 2008. DOI: 10.5170/CERN-2008-008.70.
- [297] C. Troncon. 'Detailed studies of the ATLAS pixel detectors'. In: *1999 IEEE Nuclear Science Symposium. Conference Record. 1999 Nuclear Science Symposium and Medical Imaging Conference*. Vol. 1. 1999, pp. 125–132. DOI: 10.1109/NSSMIC.1999.842461.
- [298] T. Rohe. 'Design and test of pixel sensors for the ATLAS pixel detector'. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 460.1 (2001), pp. 55–66. ISSN: 0168-9002. DOI: 10.1016/S0168-9002(00)01096-2.
- [299] Maurice Garcia-Sciveres. *RD53B Design Requirements*. Tech. rep. Geneva: CERN, 2019. URL: <https://cds.cern.ch/record/2663161>.
- [300] Jon Louis Bentley. 'Multidimensional binary search trees used for associative searching'. In: *Commun. ACM* 18.9 (July 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007.
- [301] Joana Niermann et al. *detray – a library for a GPU tracking geometry*. In preparation for submission to the Journal of Instrumentation. 2024.
- [302] Paul Gessinger-Befurt, Andreas Salzburger and Joana Niermann. 'The Open Data Detector Tracking System'. In: *Journal of Physics: Conference Series* 2438.1 (Feb. 2023). DOI: 10.1088/1742-6596/2438/1/012110.
- [303] S. Agostinelli et al. 'GEANT4 – a simulation toolkit'. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506.3 (2003), pp. 250–303. ISSN: 0168-9002. DOI: 10.1016/S0168-9002(03)01368-8.
- [304] Calafiura, Paolo, Esseiva, Julien, Ju, Xiangyang, Leggett, Charles, Stanislaus, Beojan and Tsulaia, Vakho. 'Towards a distributed heterogeneous task scheduler for the ATLAS offline software framework*'. In: *EPJ Web of Conferences* 295 (2024). DOI: 10.1051/epjconf/202429503041.
- [305] Jeronimo Castrillon, Karol Desnos, Andrés Goens and Christian Menard. 'Dataflow Models of Computation for Programming Heterogeneous Multicores'. In: *Handbook of Computer Architecture*. Ed. by Anupam Chattopadhyay. Singapore: Springer Nature Singapore, 2022, pp. 1–40. ISBN: 978-981-15-6401-7. DOI: 10.1007/978-981-15-6401-7_45-1.
- [306] Veronika Rehn-Sonigo. 'Multi-criteria Mapping and Scheduling of Workflow Applications onto Heterogeneous Platforms'. PhD thesis. Ecole Normale Supérieure de Lyon, July 2009. URL: <https://theses.hal.science/tel-00424118>.
- [307] Yi-Hsuan Kao, Bhaskar Krishnamachari, Moo-Ryong Ra and Fan Bai. 'Hermes: Latency optimal task assignment for resource-constrained mobile computing'. In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. 2015, pp. 1894–1902. DOI: 10.1109/INFOCOM.2015.7218572.
- [308] Bernard Kolman and Robert E Beck. *Elementary linear programming with applications*. Gulf Professional Publishing, 1995.
- [309] Eduardo Cuervo et al. 'MAUI: making smartphones last longer with code offload'. In: *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. MobiSys '10. San Francisco, California, United State of America: Association for Computing Machinery, 2010, pp. 49–62. ISBN: 9781605589855. DOI: 10.1145/1814433.1814441.

- [310] Tomasz Boiński and Paweł Czarnul. ‘Optimization of Data Assignment for Parallel Processing in a Hybrid Heterogeneous Environment Using Integer Linear Programming’. In: *The Computer Journal* 65.6 (Feb. 2021), pp. 1412–1433. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxaa187.
- [311] Anne Benoit, Alexandru Dobrila, Jean-Marc Nicod and Laurent Philippe. ‘Mapping workflow applications with types on heterogeneous specialized platforms’. In: *Parallel Computing* 37.8 (2011). Follow-on of ISPD’2009 and HeteroPar’2009, pp. 410–427. ISSN: 0167-8191. DOI: 10.1016/j.parco.2010.12.001.
- [312] L. R. Ford and D. R. Fulkerson. ‘Maximal Flow Through a Network’. In: *Canadian Journal of Mathematics* 8 (1956). DOI: 10.4153/CJM-1956-045-5.
- [313] Christos H. Papadimitriou. ‘On the complexity of integer programming’. In: *J. ACM* 28.4 (Oct. 1981). ISSN: 0004-5411. DOI: 10.1145/322276.322287.
- [314] ATLAS, Collaboration. *Technical Design Report for the Phase-II Upgrade of the ATLAS Trigger and Data Acquisition System – Event Filter Tracking Amendment*. Tech. rep. CERN, 2022. DOI: 10.17181/CERN.ZK85.5TDL.
- [315] Benjamin Huth. ‘Track reconstruction for future high-energy-physics experiments with classical and machine-learning methods’. PhD thesis. University of Regensburg, Mar. 2024. DOI: 10.5283/epub.55604.
- [316] Amy Brand, Liz Allen, Micah Altman, Marjorie Hlava and Jo Scott. ‘Beyond authorship: attribution, contribution, collaboration, and credit’. In: *Learned Publishing* 28.2 (2015), pp. 151–155.