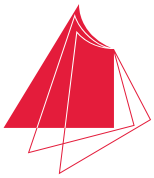


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Adding Cross-Platform Support to a High-Throughput Software Stack and Exploration of Vectorization Libraries

Master Thesis

von

Laura Promberger

am

Institute of Materials and Processes
der
Hochschule Karlsruhe – Technik und Wirtschaft

Erstgutachter:	Prof. Dr. rer. nat. Britta Nestler
Betreuender Mitarbeiter:	DI Dr. techn. Niko Neufeld

Bearbeitungszeit: September 2017 – März 2018

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die hier vorgelegte Master Thesis selbstständig und ausschließlich unter Verwendung der angegebenen Literatur und sonstigen Hilfsmittel verfasst habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde zur Erlangung eines akademischen Grades vorgelegt.

Genf, den 28. März 2018

Laura Promberger

Acknowledgments

Special thanks to Niko Neufeld for supervising my work, to Marco Clemencic for all his help solving the tricky problems within the LHCb stack and to Aritz Brosa Iartza for taking care that the machines are running. Thanks to Christian and Tommaso for taking me in as a refugee. And additionally, thanks Christian, Tommaso, Rainer, Daniel and Jon for all those very delighting discussions. Further thanks belong to Ben and Stefan (and again Niko) for guiding the general direction of this work. And last but not least, thanks to everyone who proofread this thesis.

Contents

Abstract 1

Zusammenfassung 2

1. Introduction 3
1.1. CERN . . . 4
1.2. LHC . . . 4

2. LHCb Experiment 6
2.1. The Detector . . . 6
2.1.1. The Sub-Detectors . . . 6
2.2. Data Acquisition . . . 8
2.2.1. The Software Stack . . . 8

3. Fundamentals of Vectorization 11
3.1. Horizontal vs. Vertical Vectorization . . . 11
3.2. Common Vectorization Intrinsic Sets . . . 13

4. Performance of Vectorization Libraries 14
4.1. Overview . . . 14
4.2. Libraries Selected for the Performance Analysis . . . 16
4.2.1. Vcl . . . 16
4.2.2. UMESIMD . . . 17
4.3. Performance Analysis . . . 18
4.3.1. Cross Kalman . . . 18
4.3.2. Presenting the Results . . . 24

5. Porting the LHCb stack to ARM 27
5.1. Compiling the Stack with LCG 91 . . . 27
5.1.1. LCG . . . 27
5.1.2. Gaudi . . . 28
5.1.3. LHCb . . . 28
5.1.4. Lbcom . . . 30
5.1.5. Rec . . . 32
5.1.6. Brunel . . . 33
5.2. Compiling the Stack with LCG 92 . . . 33
5.2.1. LCG 92 . . . 33
5.2.2. Problems . . . 33
5.3. Validation . . . 37
5.4. Performance . . . 38
5.4.1. Problems . . . 38
5.4.2. Results . . . 39

6. Porting the LHCb Stack to PowerPC **44**
6.1. Compiling the Stack with LCG 92 44
6.1.1. LCG 44
6.1.2. LHCb Stack 45
6.2. Validation 46

7. Summary **47**

8. Outlook **48**

A. Appendix **49**
A.1. Missing Dependencies on the ARM Platform 49

Bibliography **51**

Abstract

This master thesis is written at the LHCb experiment at CERN. It is part of the initiative for improving software in view of the upcoming upgrade in 2021 which will significantly increase the amount of acquired data. This thesis consists of two parts. The first part is about the exploration of different vectorization libraries and their usefulness for the LHCb collaboration. The second part is about adding cross-platform support to the LHCb software stack. Here, the LHCb stack is successfully ported to ARM (aarch64) and its performance is analyzed. At the end of the thesis, the port to PowerPC(ppc64le) awaits the performance analysis. The main goal of porting the stack is the cost-performance evaluation for the different platforms to get the most cost efficient hardware for the new server farm for the upgrade.

For this, selected vectorization libraries are extended to support the PowerPC and ARM platform. And though the same compiler is used, platform-specific changes to the compilation flags are required. In the evaluation of the port to ARM, the cost-performance analysis favors the tested Intel machine. Future steps will be to analyze the performance of the port to PowerPC and to improve the cross-platform support for selected vectorization libraries. The long-term goal is adding cross-platform support to the LHCb stack.

Zusammenfassung

Diese Masterarbeit ist in der Arbeitsgruppe des LHCb Experiments am CERN geschrieben worden. Im Hinblick auf die nächste Ausbaustufe des Experiments in 2021, die einen deutlichen Anstieg der zu verarbeitenden Daten bedeutet, ist sie Teil der Initiative die Software anzupassen. Diese Arbeit enthält zwei Abschnitte. Der erste Abschnitt bewertet mehrere Vektorisierungsbibliotheken und ihren Nutzen für das LHCb. Der zweite Abschnitt behandelt die Erweiterung des LHCb Softwarestacks um plattformübergreifend arbeiten zu können. In dieser Arbeit wurde der Softwarestack erfolgreich so erweitert, dass er auf der ARM-Plattform (aarch64) läuft und die Leistung mit Maschinen der aktuellen Server Farm verglichen werden konnte. Die Unterstützung von PowerPC (ppc64le) erwartet am Ende dieser Arbeit die Leistungsanalyse. Das Ziel dieser Arbeit ist die Kosten-Leistung Optimierung für die neue Server Farm für das Upgrade. Um eine größtmögliche Auswahl an Angeboten zu haben, ist die Unterstützung von mehreren Plattformen hilfreich.

Während der Arbeit wurden einige Vektorisierungsbibliotheken erweitert, sodass sie ARM und PowerPC unterstützen. Und obwohl der gleiche Compiler genutzt wurde, mussten Änderungen an den Compiler Flags vorgenommen werden um auf anderen Plattformen laufen zu können. In der Kosten-Leistungsanalyse liegt der getestete x86 Prozessor zum derzeitigen Stand vor dem ARM Prozessor. Nach der Arbeit sind folgende nächste Schritte geplant: die Leistungsanalyse der Unterstützung von PowerPC und die Verbesserung der Plattformunabhängigkeit der Vektorisierungsbibliotheken, so dass auf lange Sicht die Plattformunabhängigkeit des LHCb Softwarstacks gegeben ist.

1. Introduction

The Standard Model of particle physics is one of the most complete theories describing the interaction of particles on sub-atomic level. In particular it classifies elementary particles and describes their interactions with three of the four known fundamental forces: the electromagnetic, weak and strong force. Gravitation as the fourth force is not described by this model.

The Standard Model, as a theory of nature, is still incomplete. For example, aside from the missing gravitational force in the equation, it also does not explain the extent of the asymmetries between matter and antimatter observed in the universe. These asymmetries describe the universe having more matter than antimatter, which is crucial for the existence of our visible universe as we know it.

At CERN, the Standard Model is researched. The primary target of the Large Hadron Collider (LHC) and its experiments is the research of new physics. For this, protons or lead ions are collided at temperatures similar to temperatures shortly after the Big Bang. The particles and their decaying derivatives generated by the collision are measured. Everything not described by the Standard Model is of interest, as it would be new physics and guide extending the model.

Beginning in 2021, the LHC and its experiments will have a big upgrade with improvements in the following years. For example, the peak luminosity which is the amount of particles colliding in the same time frame will be increased by a factor of five. The increase will happen step-by-step until it is completed in 2025. As a result the collected amount of data will increase proportionally. This requires heavy refactoring of hardware and software to cope with the increasing amount of data, including improved scheduling and parallelization of the data flow and the increased use of vectorization [8].

The goal of this master thesis can be divided into two sections. The first one explores multiple vectorization libraries evaluating their usefulness, long term maintainability and cross-platform support. The second one is about adding cross-platform support to the high-throughput software stack of the LHCb experiment.

The structure of this thesis is the following: In this chapter CERN and the LHC are introduced. Then, in chapter 2 the LHCb experiment, its collaboration, its detector and software stack are presented. This is followed by a short introduction of the fundamentals of vectorization in chapter 3. Chapter 4 is all about vectorization libraries and their performance. And afterwards two chapters deal with adding the cross-platform support to the software stack. For this, chapter 5 describes the steps necessary to port the software stack to ARM (aarch64) and chapter 6 describes the first steps taken to port to PowerPC (ppc64le). The thesis is concluded by an evaluation in chapter 7 and an outlook for future steps in chapter 8.

1.1. CERN

Founded in 1954, the European Organization for Nuclear Research is a collaboration of 22 member states with the goal to research particle physics. Its abbreviation dates back to the **C**onseil **E**uropéen pour la **R**echerche **N**ucléaire, a provisional body for creating a world-class fundamental physics research organization in Europe. Nowadays, the member states are not restricted to Europe. As shown in Figure 1.1, aside from member states there are other possibilities, with different levels of rights and duties, how states can collaborate with CERN [6, 4].

CERN employs about 3,300 people and has more than 13,500 visting scientists from over 70 countries each year [31].

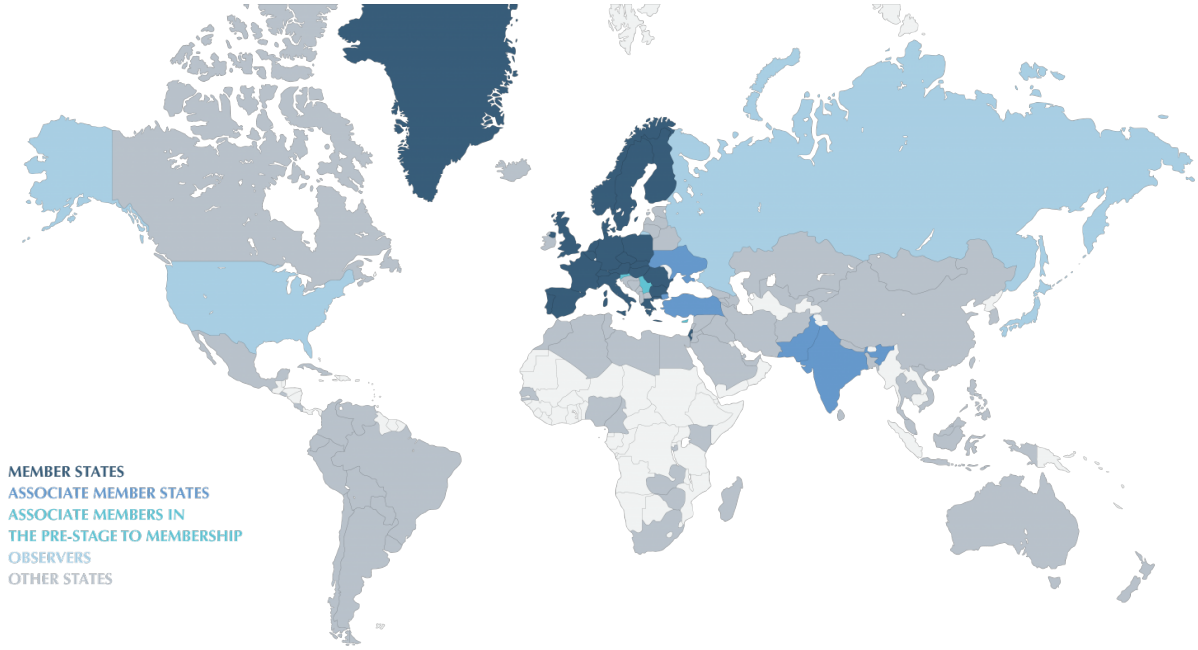


Figure 1.1. World map of the relationship between CERN and different countries [5]

1.2. LHC

The Large Hadron Collider is currently not only the largest particle accelerator, but also the largest machine on Earth. It is a synchrotron - a circular accelerator - with a circumference of 27 km and has a power consumption of 120 MW (1/3 of Canton Geneva). The LHC uses supra-conducting magnets to boost and bend bundles of particles to the right trajectory. Within the accelerator particles circulate with velocities of near light speed in opposite directions. On collision point, where the experiments are housed, magnets focus and compress the bundles to have a high probability of collisions. As seen in Figure 1.2, the LHC uses its smaller, previous accelerators to boost the particle beam up to the energy level at which the LHC magnets are able to hold the particles on the right trajectory. The LHC has an average frequency of 30 MHz - meaning that every 33 nanoseconds particle bundles collide [8, 7].

The name "hadron collider" is rooted in one of the four classes of the elementary particles. The classes are: leptons, gauge bosons, scalar bosons and quarks. The most known particles

of leptons are the electron and muon, of the gauge bosons it is the photon, and of the scalar boson it is the famous Higgs. Different quarks combined create hadrons, like neutrons and protons. The Large Hadron Collider accelerates either protons or lead ions, thus its name hadron collider.

The LHC houses seven experiments. The four largest ones are: A Large Ion Collider Experiment (ALICE), A Toroidal LHC ApparatuS (ATLAS), the Compact Muon Solenoid (CMS) and the Large Hadron Collider beauty (LHCb) experiment. Each of them is installed in their own underground cavern and has, as collaborators, between 1,200 and 3,500 members of many different nations. The three smallest experiments are: the TOTal Elastic and diffractive cross section Measurement (TOTEM), the Large Hadron Collider forward (LHCf) and the Monopole and Exotics Detector at the LHC (MoEDAL) [8].

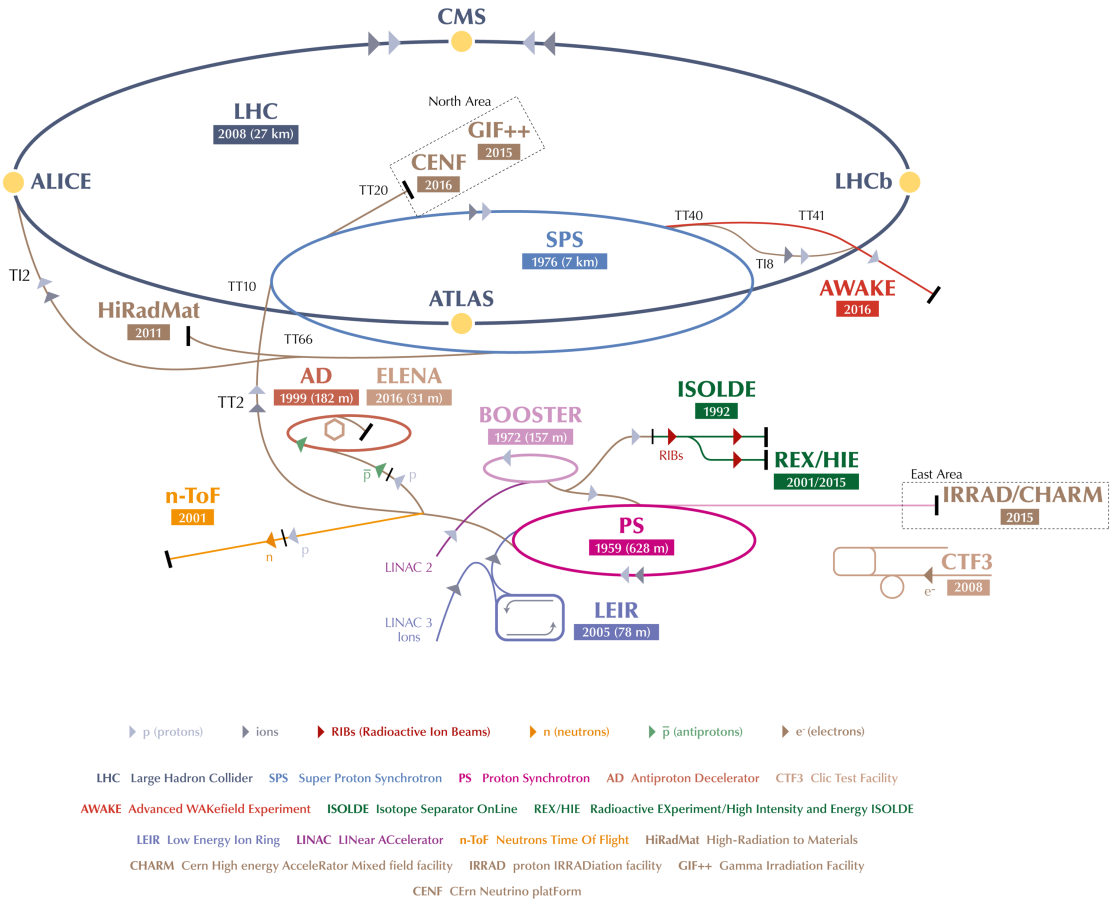


Figure 1.2. The CERN accelerator complex[30]

2. LHCb Experiment

The LHCb experiment is one of the four large experiments at CERN. Its goal is the analysis of heavy flavour particle decays, e.g. b-mesons. B-mesons consist of b-quarks and are created by proton-proton collisions. The bottom quark, also called *beauty quark* is the name giver of the experiment. As with many other experiments at CERN, the big goal of LHCb is the analysis of the matter anti-matter asymmetries. In particular, LHCb analyzes the asymmetries of the b-quarks and their decay rates that are not closely enough defined by the Standard Model. This collaboration has more than 1,200 members from 71 institutes in 16 countries [8, 14].

2.1. The Detector

The LHCb detector is built along the beam line. Beginning at the collision point it covers the collision only in one direction - meaning it only analyzes half of the collision. This is possible due to b-mesons being created in only a small cone-shaped area in the forward direction. The backward direction does not need to be measured as it can implicitly be reconstructed. The detector is 21 m long, 10 m high, 13 m and weighs 5,600 tonnes. The original construction costs were 75 MCHF.

To analyze collisions efficiently a trigger is applied. This trigger works on so-called *events*. In this case an event can be thought of as a 3-dimensional image of the collision which is acquired by the sum of all sensor data originating from the collision of two primary particles. The event trigger discards many of the already known particle interactions. To analyze the efficiency of the trigger, some randomly selected events are saved without filtering. The event trigger is composed of the hardware trigger (L0) and the software-based High Level Trigger (HLT). The HLT is responsible for further filtering and the reconstruction of the physics measured by the detectors (e. g. the particle trajectory). This near real-time computation is called *Online*. Later on, in *Offline* computing, a more precise reconstruction is used to analyze the collisions [8, 11].

2.1.1. The Sub-Detectors

As shown in Figure 2.1, the LHCb Detector is made up of multiple sub-systems, specialized for either tracking the trajectory, particle identification or measurement of particle momenta and kinetic energy. The goal is the identification of all particles and their decays.

Tracking the trajectory is done with the following sub-systems: Vertex Locator (VELO), Silicon Tracker (ST), Outer Tracker (OT) and muon detector. The VELO is the first detector and very close to the beam line and the collision point. It has a precision of about 10 μm and detects decays of particles resulting in beauty quarks. The ST detects low-momentum particles being bent out of the detection area by the magnet before reaching the next detector. The OT is the main tracker, allowing an accurate measurement of the particle's trajectory.

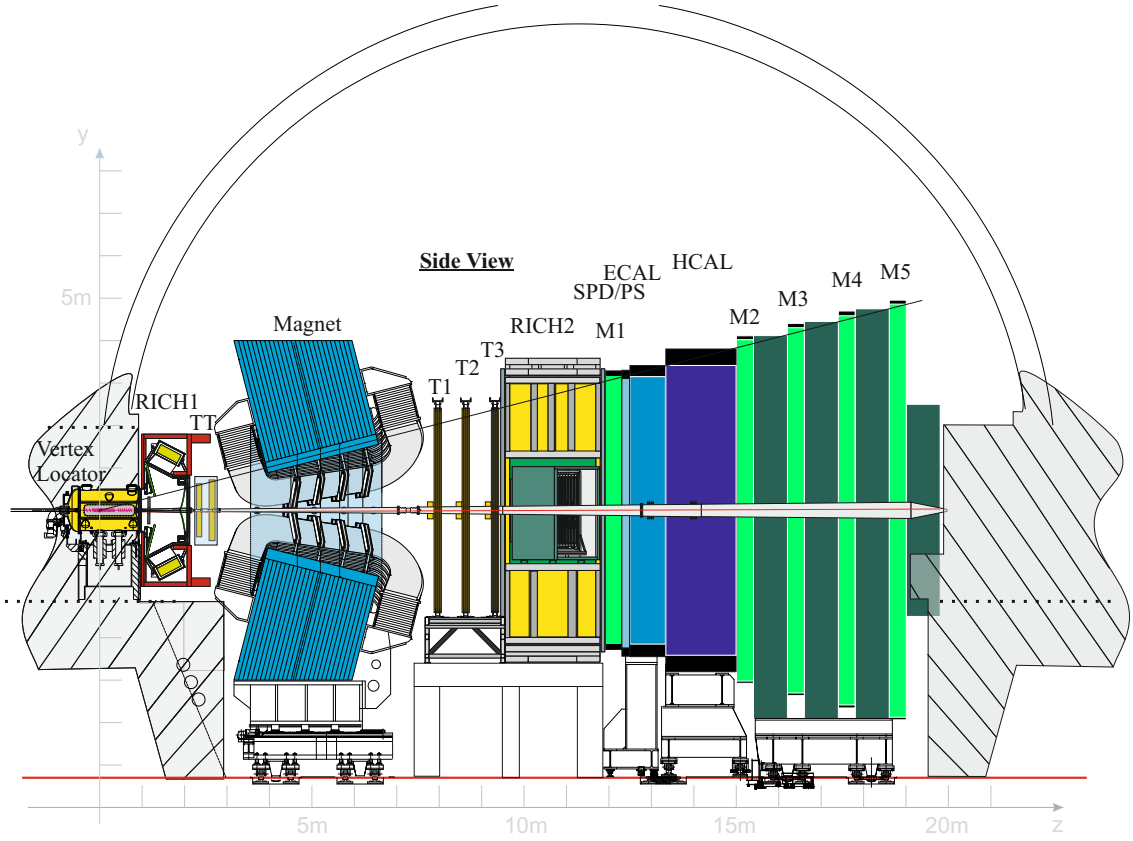


Figure 2.1. Side view of the LHCb Detector[9]

The momentum of the particle is calculated through the deviation applied by the magnetic field. The magnet has a field strength of 4 Tm and weighs 27 tons. The particle's derivation results in a helix-like trajectory and depends on both components of the momentum: the particle's velocity and mass.

The kinetic energy is measured using calorimeters. The sub-systems for this are: PreShower (PS), Scintillator Pad Detector, Electromagnetic Calorimeter (ECAL) and Hadron Calorimeter (HCAL). Different calorimeters are used to analyze different type of particles (electrons, photons, hadrons, etc.). In all cases it is a destructive process. On impact, the particle is destroyed but creates a shower of charged particles. These showers create so-called scintillating light which is proportional to the energy of the incoming particle and can be measured.

The sub-systems for the particle identification are two Ring Imaging Cherenkov detectors (RICH-1 and -2), the calorimeters and the muon detector. The RICH detectors are filled with material, in which charged particles emit cones of so-called Cherenkov photons. Cherenkov photons are light emitted by objects travelling faster than light speed through a specific medium. Depending on the velocity of the particle, the angle of a cone differs. Knowing the trajectory and momentum allows a prediction of the velocity. With this it is possible to determine the mass - and thus ultimately identify the particle. The muon detector is the last detector in the whole system. Muons are important as they are part of many B-meson decays. They have a very weak interaction with matter. To get a precise measurement, all other particles but the muon are destroyed before by the calorimeters. Furthermore, to prevent misidentification, there

are iron blocks in between the muon detection-layers to filter out missed non-muon particles [8, 11].

2.2. Data Acquisition

The data acquisition in Online and Offline is done by a common software stack. However, the algorithms used differ in speed and accuracy, depending on the requirements of Online and Offline. Currently, for Run 2 (2015 - 2018), LHCb has a data acquisition rate of 50 GB/s from the detector and a data recording rate of about 1 GB/s between the hardware trigger L0 and software trigger HLT.

With the upgrade and the discontinuation of the hardware trigger, the data acquisition rate is going to increase to 4 TB/s and the data recording rate is going to be approximately 10 GB/s.

2.2.1. The Software Stack

The LHCb software stack is comprised of multiple large projects which are comprised of a total of 5 million lines of code. It is mainly written in C++ and uses Python scripts for configuration. The first version of it was developed over ten years ago. For this master thesis, the following projects are of interest: LCG, Gaudi, LHCb, Lbcom, Rec, Brunel. The dependencies between the projects are shown in Figure 2.2.

LCG

LCG is a software bundle used by all LHC experiments. It provides the infrastructure for the software stacks used by the different experiments. Each version of LCG contains a compatible set of (external) software packages. The software packages offered are from many different areas such as

- General tools (debugging, testing)
- Graphics
- Mathematical Libraries
- Databases
- Scripting Languages and modules
- Grid middle ware
- Compilers

LCG is based on the ExternalProject module of CMake and is maintained by EP-SFT, the department for SoFTware Development for Experiments [12].

Gaudi

Gaudi is an experiment-independent framework which is "usable for implementing the full range of offline computing tasks: the generation of events, simulation of the detector, event reconstruction, testbeam data analysis, detector alignment, visualisation, etc. etc.."[1]. Furthermore it also supports the implementation of online computing tasks. Gaudi allows the execution of an algorithm by Gaudi itself or by other algorithms. This allows a partitioning of complex tasks [3].

Gaudi is jointly developed with Atlas and uses LCG as its dependencies provider.

LHCb

The LHCb project is the experiment-specific project built on top of Gaudi. It contains the event model, detector description and interfaces. The event model describes the data types used for each part in the trigger and an analysis and of their relationships. The detector description is divided into structure, geometry and materials of which the detector is composed. This is described in XML files [24, 25].

Lbcom

The next project in the software stack is Lbcom. It is built on top of LHCb and contains modules which are commonly used by multiple data processing projects that follow in the software stack [26].

Rec

Rec is the project containing all relevant algorithms for the reconstruction of the particle tracks. It uses Lbcom [27].

Brunel

Brunel is the last project used for this LHCb stack configuration. It is the user interface, allowing to specify job options and run the algorithms defined in Rec and Lbcom. Brunel can be used for both real data provided by the detector and for Monte Carlo simulations provided by the project Boole. The option files are Python scripts [23].

Other Projects

Other projects used by LHCb include DIRAC and Ganga. DIRAC is an interware, transparently scheduling and load balancing tasks on distributed resources. This includes catalog, storage and computing resources. The LCHb Grid uses an extension of DIRAC, called LHCbDIRAC [36, 35]. The other example, Ganga, is a frontend tool written in Python. It allows to compose, run and track compute jobs. It is developed by ATLAS and LHCb and optimized for using Gaudi or Athena as a framework [40].

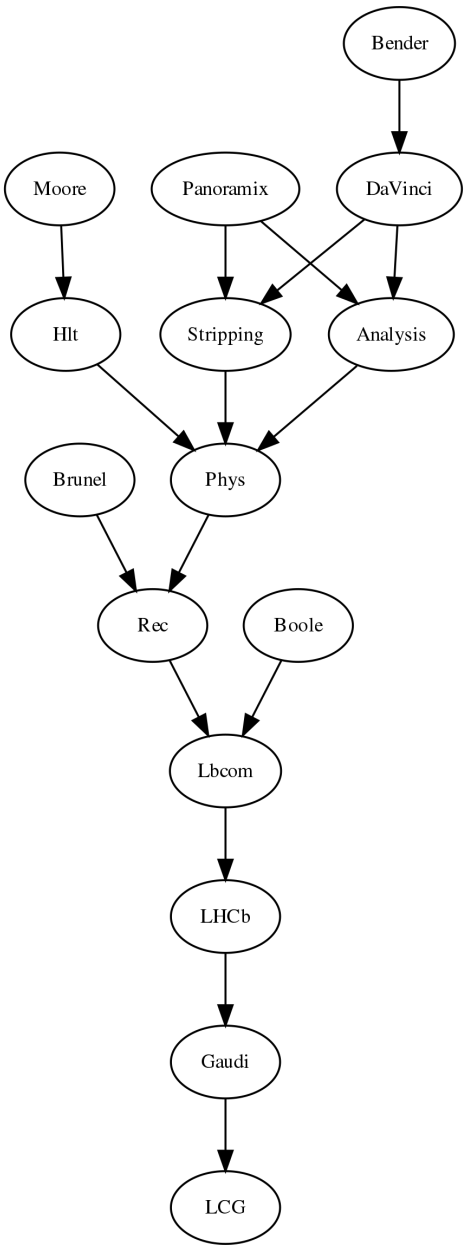


Figure 2.2. LHCb Stack: Graph of the dependencies between projects [22]

3. Fundamentals of Vectorization

In Computer Science the so-called *Flynn's taxonomy* defines four different classes for computer architectures: single-instruction stream – single-data stream (SISD), single-instruction stream – multiple-data stream (SIMD), multiple-instruction stream – single-data stream (MISD), multiple-instruction stream – multiple-data stream (MIMD). Ultimately, it describes the relationship between data and commands within the CPU (see Table 3.1) [15].

In early time the common computer was only able to process SISD. One instruction would manipulate one chunk of data at one time. However, compared to SIMD or MIMD, that manipulate multiple data chunks at the same time, SISD is inefficient. Nowadays, modern computers support, aside from the traditional SISD, also SIMD. SIMD allows the execution of one instruction on multiple data chunks in almost the same time as compared to SISD. These special SIMD-commands are *intrinsic* functions, which the compiler substitutes by one or multiple assembler operations. The process of implementing a SIMD-using program is called *vectorization*. The data type on which SIMD intrinsics work is called *vector*. The vector width depends on the SIMD register widths.

Table 3.1. Flynn's Taxonomy

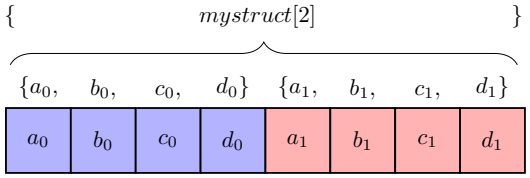
	Single-Instruction Stream	Multiple-Instruction Stream
Single-Data Stream	SISD	MISD
Multiple-Data Stream	SIMD	MIMD

3.1. Horizontal vs. Vertical Vectorization

There are two styles of using vectorization on data, vertical and horizontal vectorization. Depending on which type is used, the data structures have to be adjusted to maximum performance. The preferred data structure for scalar and vertical vectorization is *Array of Structs* (AoS). For horizontal vectorization it is *Struct of Arrays* (SoA). AoS is the traditional data representation. It describes having one array of multiple objects with each having its own private data - the array is partitioned by the objects. Whereas the design of SoA is an array partitioned by the parameters. As the objects are all of the same type, they have the same number of parameters. Hence in SoA, the array will consist of the first parameter of all objects, then the second parameter of all objects, and so on. Having the correct data representation for the used vectorization style allows a sequential load for the given vectors. This significantly reduces the loading time of a vector due to caching.

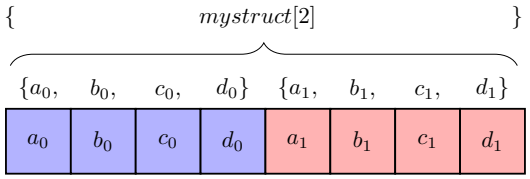
As an example Figure 3.1 shows a scalar implementation of a simple computation (two additions for each data structure, blue and red). Afterwards the same computation is done using vertical and horizontal vectorization and their preferred data structure that benefits from the sequential load of vectors. The vector size is two and a vector is represented by $< value0, \quad value1 >$.

(a) Scalar with AoS



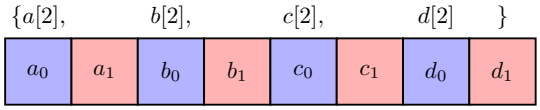
$$x_0 = a_0 + c_0$$
$$y_0 = b_0 + d_0$$
$$x_1 = a_1 + c_1$$
$$y_1 = b_1 + d_1$$

(b) Vertical Vectorization with AoS



$$< x_0, y_0 >=< a_0, b_0 > + < c_0, d_0 >$$
$$< x_1, y_1 >=< a_1, b_1 > + < c_1, d_1 >$$

(c) Horizontal Vectorization with SoA



$$< x_0, x_1 >=< a_0, a_1 > + < c_0, c_1 >$$
$$< y_0, y_1 >=< b_0, b_1 > + < d_0, d_1 >$$

Figure 3.1. Different vectorization types and their preferred data structure

The horizontal vectorization offers the advantage of being much closer to the scalar representation. However, to be efficient, often the data structures have to be rearranged to allow a sequential load of the vectors. The preferred data arrangement for the horizontal vectorization is SoA and AoS for the vertical vectorization.

In Table 3.2 is a summary of the advantages and disadvantages using either horizontal or vertical vectorization.

Table 3.2. Horizontal vs. Vertical Vectorization

Vectorization Style	
Horizontal	Vertical
use SoA	use AoS
often transformations AoS → SoA → AoS are necessary	most programs use AoS data structures
math looks like the scalar version	complex rearranging of the math is necessary
better scaling	hard to change because of dependencies within the math

3.2. Common Vectorization Intrinsic Sets

Using intrinsic sets for vectorization is a low-level and hardware-dependent approach. Depending on the platform type and the exact processor different intrinsic sets with different register sizes are supported. In Table 3.3 the most common intrinsic sets are listed which are used on platforms relevant in this thesis.

Table 3.3. Common vectorization intrinsic sets

Platform	Intrinsic Set	Register Size (Bits)	Vector size of `double`
Intel	SSE (up to SSE4.2)	128	2
Intel	AVX, AVX2	256	4
Intel	AVX512	512	8
IBM (ppc64le)	Altivec	128	2
ARM (aarch64)	NEON	128	2

4. Performance of Vectorization Libraries

To achieve the required performance increase for the upcoming upgrade, LHCb increases the use of vectorization. One option is to use vectorization libraries. They allow to hide the complexity and pitfalls of architecture-dependent intrinsic implementations, and - in theory - allow the portability of the software across architectures. LHCb already uses two different vectorization libraries, Vcl and Vc.

In the following chapter four vectorization libraries Vcl, Vc, UMESIMD and boost.simd are compared. The selection is based on the general popularity of the libraries and the interests of the LHCb group. Afterwards, the focus is put on Vcl and UMESIMD, including a performance analysis. Changes and additions to these libraries made throughout this work are also presented. For further discussion the results are presented to the developers working on the LHCb upgrade.

4.1. Overview

The analysis of the vectorization libraries focuses on the support of multiple architectures and their respective intrinsic sets (as defined in Table 4.1), the best performance and their long-term maintainability. In Table 4.2 an overview of the vectorization libraries at the beginning of the research is shown (September 2017). All those libraries require the developer to align the data manually and implement padding where necessary. The colors green, yellow and red signal how good the libraries fit the requirements.

Analyzing the table has the following conclusion: boost.simd is no option for LHCb especially because of the clash of interests with bSIMD. Vc is a high-level wrapper which therefore also does not cover all vectorization use cases. For the further analysis of the cross platform support and the performance, Vc cannot be selected as it currently does not have the required architecture support. And additionally, extending it is too complex to do it alone. As a result, for further analysis Vcl and UMESIMD are selected. They have the advantage that there is already a standalone implementation of a realistic LHCb algorithm that allows to switch between Vcl and UMESIMD as back end.

Table 4.1. Requested architecture support

Company	Intrinsic Set	Register Size (Bits)
Intel	AVX2	256
Intel	AVX512	512
IBM (ppc64le)	Altivec	128
ARM (aarch64)	NEON	128

Table 4.2. Vectorization libraries

	VCL	VC	UMESIMD	BOOST.SIMD
AVX2	Yes	Yes	Yes	Yes
AVX512	Yes	In Development	Yes	No
ALTIVEC	No	No	Early Example	No
NEON	No	In Development	Early Example	No
DOCUMENTATION	Yes	Yes	Incomplete	Yes
EXAMPLES	Yes	Yes	Yes	Yes
VECTOR SIZE	Architecture-dependent	Architecture-dependent	Fixed to certain sizes, but architecture-independent	Architecture-dependent
VECTOR TYPES	Mathematical primitive data types	Mathematical primitive data types and `char`	Mathematical primitive data types	Every type which is of type vectorized and is vectorizable
MASKED FUNCTIONS	shuffle, blend and permute	Nearly every function (`where` construct)	Nearly every function	if_else for boolean functions; masked pointers
UNSUPPORTED FUNCTIONS AND ARCHITECTURES	No	Officially yes, but not implemented	Scalar emulation	Partial scalar emulation?
MISCELLANEOUS	Offers optimized Math functions	ROOT (used by LHCb) uses Vc for vectorized functions		
LICENSE	GPL3 or payed for use in proprietary software	BSD-3-Clause	MIT	Boost Software License[39]
DISTRIBUTION	own website[16]	Github[20]	Github[19]	Github[33]
MAIN DEVELOPER	Agner Fog	Matthias Kretz	Przemyslaw Karpinski	NumScale
VECTORIZATION STYLE	wrapper for intrinsic	Targeted for horizontal vectorization	wrapper for intrinsic	Vectorization of boost library
VECTORIZATION TYPE	horizontal and vertical	horizontal	horizontal and vertical	?

Table 4.2. Vectorization libraries

	Vcl	Vc	UMESIMD	boost.simd
Problems	Only support for Intel architecture, not so many masked functions	Does not cover all use cases; hard to implement vertical vectorization; vector width not always identifiable	Many functions are still emulated	Depends on boost; clash of interests: is a subset of the closed-source project bSIMD which supports more architectures
Source Files contain	all functions for one intrinsic set and multiple vector sizes and masks	multiple functions with multiple intrinsic sets and vector sizes	all functions for one intrinsic set and one vector size	stretched over multiple files, with multiple intrinsic sets and vector sizes
Extensibility for new intrinsic	medium (no unit tests)	complex	easy (unit tests available)	complex
Future		Wants to be integrated in C++ standard library	support for new ARM intrinsic set with no developer-controlled vector size (SVE)	

4.2. Libraries Selected for the Performance Analysis

4.2.1. Vcl

Vectorclass (Vcl) is developed by Agner Fog. It is a library offering highly optimized low-level wrapper for Intel intrinsic sets. It has no support for any other architectures. Aside from Vcl, Agner Fog also offers many documents about code optimization [16].

In Vcl the vector size depends on the intrinsic set being used. If, for example, a vector of size 512 bits is used the AVX512 intrinsic set must be available on the machine. There is no emulation of other vector sizes implemented.

Extension for Multi-Architecture Support

Vcl does not offer implementations either for Altivec (ppc64le) or for NEON (aarch64). Instead, as the performance should be analyzed on different architectures, Vcl has to be extended for those two platforms. Altivec and NEON have the same register width of 128 bits, therefore the needed extensions apply to the same functions.

In the beginning this extension is only designed to support the program of the performance analysis, therefore only a partial implementation of `double` and `float` is done. This includes masks and common functions like add, multiply, square root, compare operators, select, load, store and some more. However, it is important to note that those implementations do not offer a full support for all Vcl data types and functions.

Agner Fog does not offer any unit tests to the public. Nevertheless, it has to be assumed that Agner Fog's own implementation is thoroughly tested. Based on this, the verification of the new implementations is done against the results on an Intel machine.

The code with the added extensions can be found on the CERN Gitlab server: see [37].

4.2.2. UMESIMD

UMESIMD is a cross-architectural vectorization library developed by Przemyslaw Karpinski as part of a project at CERN. It supports the most common intrinsic sets of multiple architectures - though some of them are just an early example. It was inspired by Vcl and Vc [19].

UMESIMD offers a homogeneous interface for all intrinsics, independent of the underlying architecture. If a platform or an operation is not supported (or not yet implemented as intrinsics) a scalar emulation is used. UMESIMD has a masked version for nearly every reasonable function. Apart from this, it distinguishes between functions that assign the value to a different variable, and functions that assign the value to one of the input vectors. For many functions both options are available with different, optimized implementations, as shown in Code Section 4.1.

Furthermore, the vector size is not bound to the architecture being used. For example, when working on a machine which supports AVX512, the vector size does not have to be 512 bits. It could also be 256 bits or even 1024 bits. However, those unusual vector sizes might just be an emulation and not yet implemented as intrinsics. As a result, it is currently recommended to use a vector size that fills the full register width of the current machine.

```
// ABS absolut value
UME_FORCE_INLINE SIMDVec_f abs() const {
  __vector float t0 = vec_abs(mVec);
  return SIMDVec_f(t0);
}

// MABS masked absolut value
UME_FORCE_INLINE SIMDVec_f abs(SIMDVecMask<4> const & mask) const {
  __vector float t0 = vec_abs(mVec);
  __vector float t2 = vec_sel(mVec, t0, mask.mMask);
  return SIMDVec_f(t2);
}

// ABSA self-assign absolut value
UME_FORCE_INLINE SIMDVec_f & absa() {
  mVec = vec_abs(mVec);
  return *this;
}

// MABSA masked self-assign absolut value
UME_FORCE_INLINE SIMDVec_f & absa(SIMDVecMask<4> const & mask) {
  __vector float t0 = vec_abs(mVec);
```

```
  mVec = vec_sel(mVec, t0, mask.mMask);
  return *this;
}
```

Code Section 4.1 Different implementations depending on assignment and masking

Extension for Multi-Architecture Support

Similar to the extension of Vcl, UMESIMD also needs `double` and `float` extensions for both, Altivec and NEON. UMESIMD already had some exemplary implementation for some of the vector types. This helped to get started. Albeit, during the work, several improvements are done compared to the exemplary implementation.

To test the results, UMESIMD has a huge set of unit tests, testing every function for every vector type and mask. Unfortunately, the current testing has some disadvantages

1. The compile time is very long, as everything is templated and no subsets can be selected.
2. The test creates random values and calculates the expected result with scalar functions. Then, it compares it with the results of the vectorized version. However, some functions do not correctly handle `NaN` and `Infinity`, resulting in occasional failings of tests.
3. The unit tests do not cover every corner case. They only test a random subset of all possibilities.

4.3. Performance Analysis

For the performance analysis Vcl and UMESIMD are selected. Both libraries have a similar interface which reduces the complexity and workload of preparing for the analysis. Furthermore, it also reduces the bias when evaluating the performance as the replacement is minimalistic. The project being chosen is `cross kalman`.

4.3.1. Cross Kalman

`cross kalman` is a cross-architectural Kalman filter developed by Daniel Hugo Cámpora Pérez [2]. As backend it supports scalar and vectorized computation. The vectorization backend allows to switch between UMESIMD and Vcl.

Kalman Filter

The Kalman filter was invented by Rudolf E. Kálmán in 1960. It is a data fusion algorithm, allowing to calculate a new state based on the previous state and some corrective value, e. g. corrections due to external influences. A famous early time use of the Kalman filter was in the Apollo navigation system. Its function was to estimate and correct the trajectory while the manned spacecraft traveled around the moon. Meanwhile considering important factors, like wind, thrust and break maneuvers, moon speed and the descent angle [17].

In general, the Kalman filter uses a theoretical model to predict the new value, and the corrective value is measured. Both, the model and the corrective value are represented as Gaussian probability density functions (PDFs) - enabling to introduce uncertainties and noise. Afterwards, the product of those two PDFs is created. The peak of the product is the most likely value of the new state. One advantage of the Kalman filter is its good recursive implementation. A unique property of Gaussian PDFs is that a product of two Gaussian PDFs will always result in another Gaussian PDF. This allows an efficient recursive implemention of the Kalman filter [13].

At LHCb, the Kalman filter is used at multiple stages of the track reconstruction. Its objective is to remove outliers or disregard entire tracks, so-called ghost tracks, due to high Chi-square (χ^2) values. `cross kalman` is a realization of the fitter used at the Fast and Best selection stage of the HLT, where tracks are fitted forward, backward, and averaged in a process called smoothing. The smoothed trajectory then is used to classify tracks, identify ghosts and keep the best track candidates.

Performance

The performance analysis was performed on the machines listed in Table 4.3. Here, the column NUMA Nodes describes the memory layout of the machine.

Non-Uniform Memory Access (NUMA) is a concept of multi-core machines with a shared memory layout. Here, each CPU has an own memory controller and one part of memory is directly connected to it. Such a group of CPUs and memory mounted on a socket is called NUMA node. The NUMA approach reduces the bottleneck of the access time compared to the Uniform Memory Access (UMA) design. In UMA there is only one RAM location and to access it each CPU has to request it through the Memory Controller Hub. However compared to UMA, the NUMA design introduces locality of the memory access. Meaning, the access time within the NUMA node is lower than accessing memory located in a different NUMA node. Thus the name Non-Uniform Memory Access. As a consequence a program will have increased performance if bound to a singled NUMA node, provided it fits into the local NUMA node RAM. Checking the memory layout of the current machine and binding a program to a NUMA node can be done with the tool `numactl` and `hwloc` [29].

To measure the performance, `cross kalman` is run several times with a different amount of cores being used. For each NUMA node, a separate program instance is started and bound to it. The test consists of a data set of 75 events, containing multiple tracks. The base number of events to be evaluated is 500,000 (re-iterating over the same data set), which is equally split by the number of NUMA nodes. For example, a machine with two NUMA nodes runs 250,000 events in each instance, whereas a machine with four NUMA nodes runs 125,000 events in each instance.

The KNL machine allows a sub-clustering of NUMA nodes. This particular KNL has a sub-NUMA clustering (SNC) of four. Each of the four NUMA instances have access to their own DRAM and MCDRAM. Multi-Channel DRAM (MCDRAM) is a special kind of high-bandwidth RAM. It can either be used as L3-Cache or as a NUMA node. On this machine there is 16 GB of MCDRAM, split onto four NUMA nodes making it 4 GB per node. `cross kalman` uses about 1.5 - 1.7 GB per instance, thus it can run solely on the high-bandwidth MCDRAM [18, 34].

The Testsystem ARMv8 cannot be fully named, as it is protected by a non-disclosure agreement (NDA). Also, it was added at a later time of the analysis, therefore some analysis sections do not include it. This also applies to the ThunderX2, which was added at an even later time.

The performance analysis of Vcl and UMESIMD was not only done on different machines, but also - if available - with different compilers. The compilers used for the machines are listed in Table 4.4. The Icc Compiler used for the Intel machines includes the Short Vector Math Library (SVML). This is an optimized vectorized math library for Intel processors. Even though Gcc has the compile flag `-mveclibabi=svml`, it was not possible to run Gcc with SVML.

Table 4.3. Machines for the performance test

Company	CPU	Cores	NUMA nodes	RAM (GB)	Largest Intrinsic Set
Intel (x86_64)	Xeon CPU E5-2630 v3 @ 2.40GHz	32	2	64	AVX2
Intel (x86_64)	Xeon Phi CPU 7210 @ 1.30GHz (KNL)	256	4 + 4 (MCDRAM)	192 + 16 (MCDRAM)	AVX512
IBM (POWER8)	Wistron Polaris Dual socket @ 3.325 GHz	128	2	267	Altivec
ARM (ARMv8)	ThunderX1 Dual socket	96	2	264	NEON (aarch64)
ARM (ARMv8)	Testsystem ARMv8	64	4	264	NEON (aarch64)
ARM (ARMv8)	ThunderX2	224	2	255	NEON (aarch64)

Table 4.4. Compilers for the performance test

Company	CPU	Compilers
Intel	Xeon CPU E5-2630 v3 @ 2.40GHz	Gcc 6.2, Gcc 7.1, Icc 18 (with SVML, backend: Gcc 6.2)
Intel	Xeon Phi CPU 7210 @ 1.30GHz (KNL)	Gcc 6.2, Gcc 7.1, Icc 18 (with SVML, backend: Gcc 6.2)
IBM	Wistron Polaris Dual socket @ 3.325 GHz	Gcc 6.2
ARM	ThunderX1 Dual socket	Gcc 6.2
ARM	Testsystem ARMv8	Gcc 7.2
ARM	ThunderX2	Gcc 7.2

New Compiler Flags

Ricardo Nobre([32]) improved the choice of the Gcc build flags. The performance increase due to the new flags is shown in Table 4.5. All following performance analysis will use these

compiling flags for the Gcc.

The old flag `-O2` was replaced with

```
-Ofast -fmodulo-sched-allow-regmoves -fno-gcse -fira-share-spill-slots
 -fmath-errno -fno-tree-dominator-opts -fconserve-stack -fno-gcse-lm
  -ftree-vectorize -fweb -fno-inline-small-functions -floop-block
-fno-merge-constants -fno-rerun-cse-after-loop -fsel-sched-pipelining
     -fno-ipa-ra -ftree-reassoc -fgcse-sm -fno-branch-count-reg
          -fno-fast-math -fno-reorder-blocks-and-partition
```

For PowerPC and ARM the flag `-floop-block` is not added, as Gcc does not support it on those platforms.

Inspired by the Gcc flag changes, following flags were used by the Icc

```
      -fimf-use-svml=true -foptimize-sibling-calls -fjump-tables
   -qopt-prefetch=3 -use-intel-optimized-headers -fma -global-hoist
          -fmerge-constants -restrict -funroll-loops -ipo
-ftls-model=local-exec -fp-trap=none -prec-div -prec-sqrt -DUME_USE_SVML
```

Table 4.5. Average performance difference: New flags speed-up compared to the old flag

Company	Gcc 6.2	Gcc 7.1	Icc18
Intel Xeon CPU E5-2630 v3 @ 2.40GHz			
UMESIMD	3%	5%	
Vcl	4%	4%	
Intel Xeon Phi CPU 7210 @ 1.30GHz (KNL)			
UMESIMD	4%	3%	141%
Vcl	4%	3%	139%
IBM Wistron Polaris Dual socket @ 3.325 GHz			
UMESIMD	8%		
Vcl	9%		
ARM ThunderX1 Dual socket			
UMESIMD	1%		
Vcl	1%		

Performance Difference: UMESIMD vs. Vcl

The average performance differences between Vcl and UMESIMD are shown in Table 4.6. Even though UMESIMD is a strongly templated library there is no significant performance difference.

Table 4.6. Average performance difference: UMESIMD speed-up compared to Vcl

Company	CPU	Gcc 6.2	Gcc 7.1	Icc 18
Intel	Xeon CPU E5-2630 v3 @ 2.40GHz	0%	1%	3%
Intel	Xeon Phi CPU 7210 @ 1.30GHz (KNL)	0%	1%	1%
ARM	ThunderX1 Dual socket	0%		
ARM	Testsystem ARMv8	-1%		
IBM	Wistron Polaris Dual socket @ 3.325 GHz	-3%		

Performance Difference: Compilers

The average performance of Vcl and UMESIMD using different compilers is shown in Table 4.7. The performance increase is compared to the compiler with the lowest throughput. In this comparison the ARM and IBM machines are excluded, as only one compiler was available.

Table 4.7. Average performance difference: Compilers (speed-up compared to the slowest one)

Library	Gcc 6.2	Gcc 7.1	Icc 18
Intel Xeon CPU E5-2630 v3 @ 2.40GHz			
UMESIMD	4%	3%	0%
Vcl	8%	6%	0%
Intel Xeon Phi CPU 7210 @ 1.30GHz (KNL)			
UMESIMD	7%	6%	0%
Vcl	8%	6%	0%

Scalability

In Figure 4.1 the scalability of the full machines is shown. It uses a logarithmic scale. The scalability plot describes how the program uses the resources compared to the increased use of cores (including hyper-threading). The perfect scalability behavior in a logarithmic scale is a straight line. However, often bottlenecks, like the limiting bandwidth of the RAM access, occur which slow down the computation.

To further analyze the scalability behavior Figure 4.2 shows the normalized scalability and Figure 4.3 shows the parallel efficiency. Both graphs contain the same information. The normalized scalability plot is interpreted in the same way as the scalability plot. The closer it is to the line, the better the scalability. The graph allows to analyze how well each machine scales compared to the others. The parallel efficiency graph has a different approach. Here a straight line at 100% is the perfect scalability. The decrease in the efficiency due to using more cores is plotted. The parallel efficiency plot is more detailed, but the understanding is not as intuitive.

The scalability graph shows that the E5-2630 v3 starts the strongest, but as it only has 32 cores it cannot keep up with the many more cores other machines have. The KNL has the best overall performance (4 NUMA nodes, MCDRAM), followed by the ThunderX2 (2 NUMA nodes). However, looking at the parallel efficiency graph shows that the ThunderX2 has the worst scalability behavior, and the KNL one of the best. Surprisingly, even though the ThunderX1 has a bad overall performance, it displays the best scalability behavior. It is to note, that the ThunderX2 is not a revision 1 of the ThunderX1, but a completely different CPU.

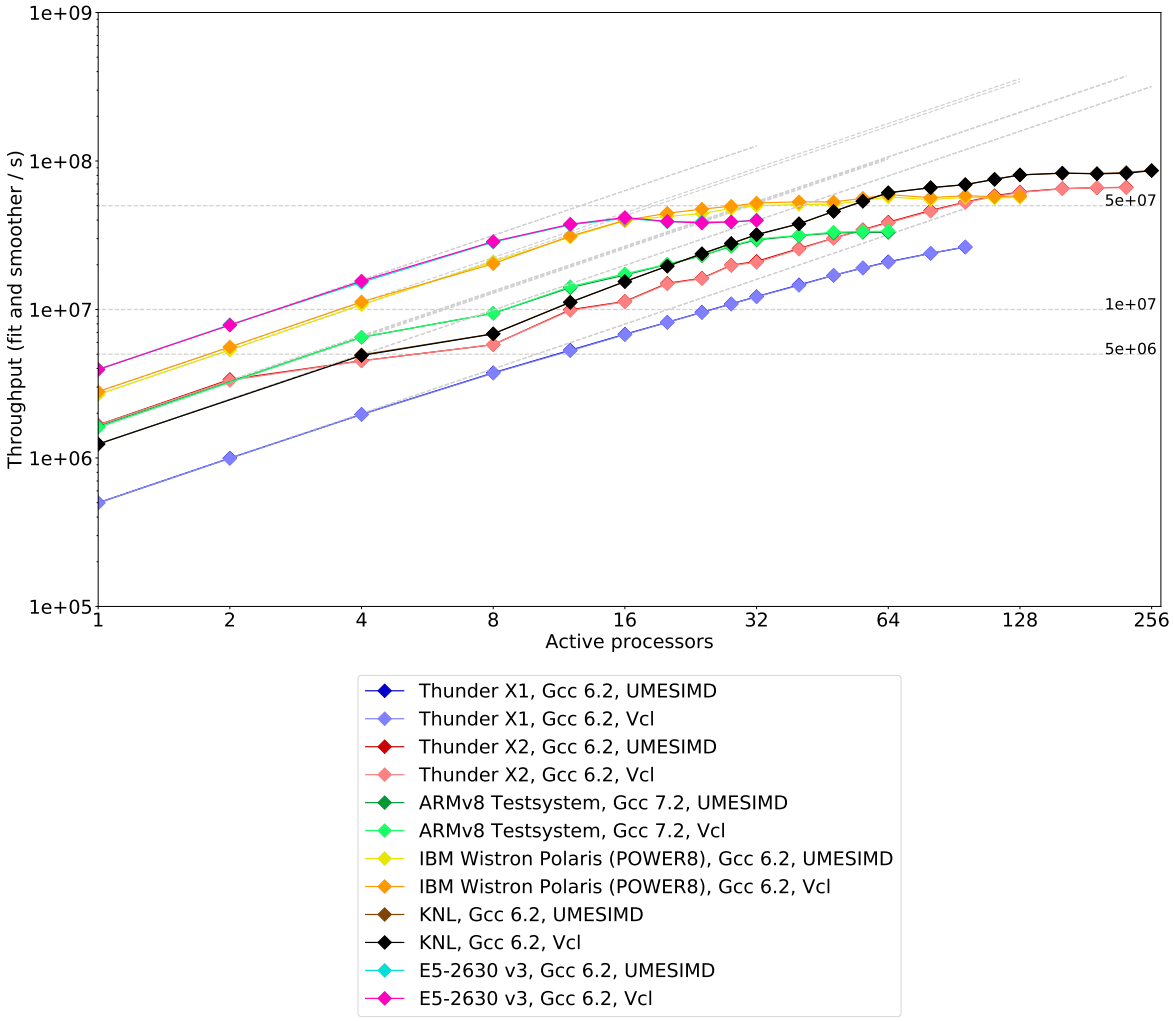


Figure 4.1. Scalability of cross kalman

Price-Performance Ratio

The price-performance ratio can be calculated by the throughput (events per second on the full machine) divided by the cost of the machine. As this price will probably be pretty low, a factor can be used, e.g. the price for 10,000 events. Only the capital expenses are taken into account here, not the running cost. So differences in power consumption do not play a role here.

However, for this analysis a price-performance comparison is not possible, as the market price of some machines is not known.

4.3.2. Presenting the Results

The results of the performance analysis were presented at the LHCb Hackathon and the LHCb Workshop. The goal was to motivate the collaboration to only use one vectorization library to improve the long-term maintainability and to encourage the collaboration to opt for a cross-platform compatible library. Though the feedback was manifold, no decision was taken. The main problem is that LHCb on its own would have to provide, test and fix new implementations for UMESIMD, as there is no paid development team for UMESIMD. For Vcl, LHCb must provide own implementations outside the Intel platform. And for Vc, LHCb does not have to provide manpower, but has to wait for it until the features are added (which can be quite a long time). Therefore, the collaboration could not decide for one vectorization library or the other.

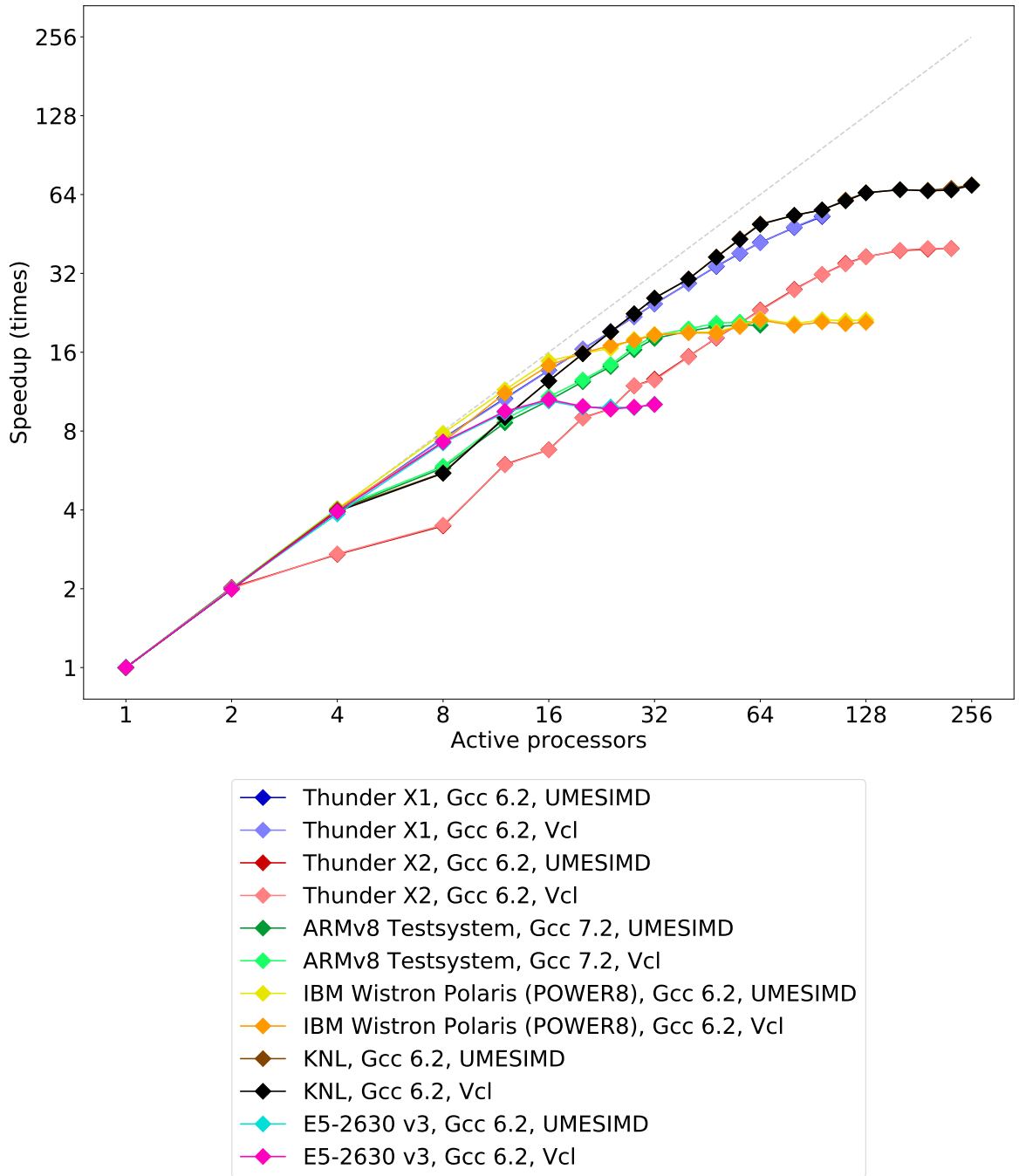


Figure 4.2. Normalized scalability graph

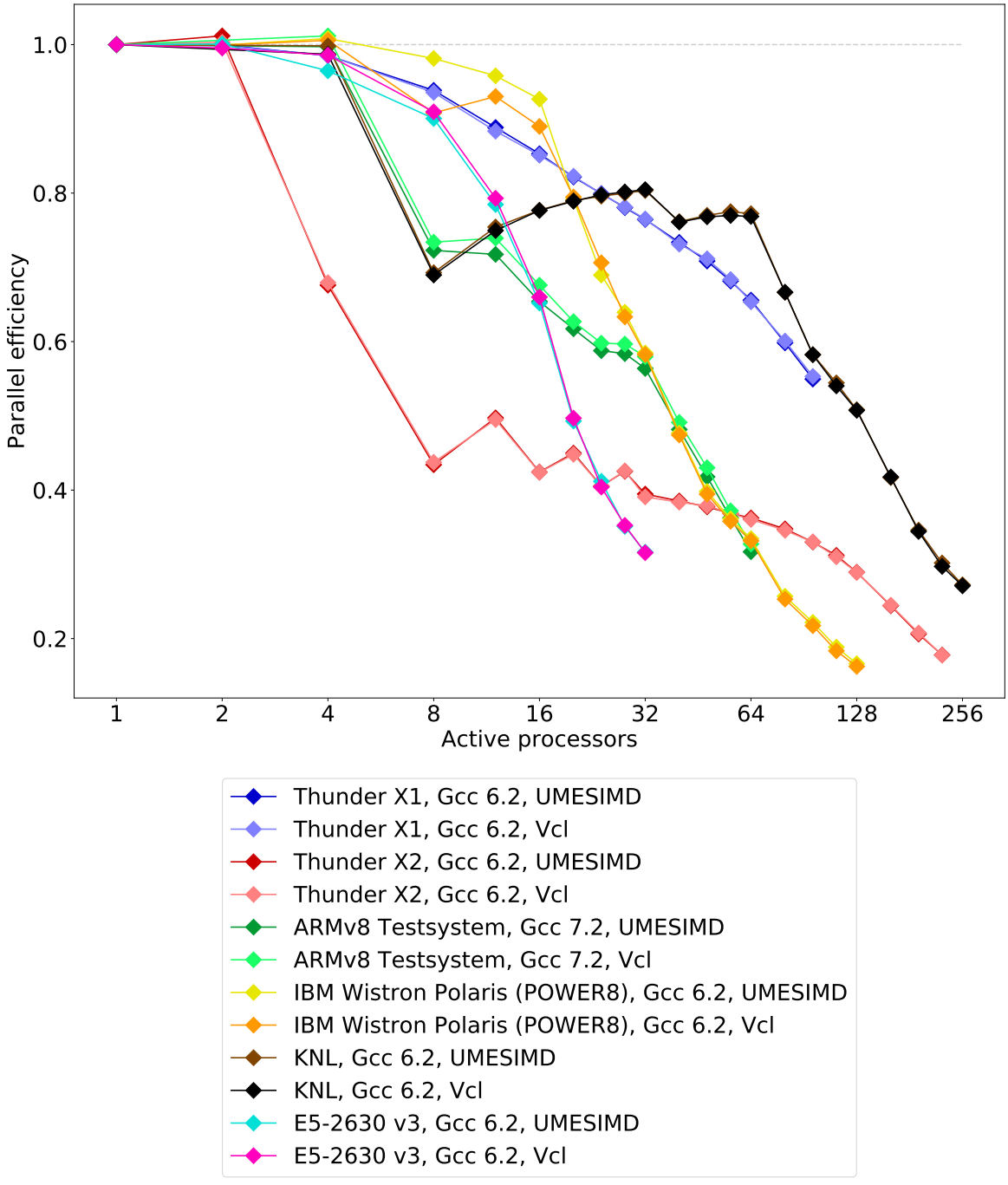


Figure 4.3. Speedup: Normalized scalability graph

5. Porting the LHCb stack to ARM

For the upcoming upgrade, LHCb will buy a new server farm with more computational power. For this, a bigger data center is built on the site where the experiment is placed. For price-cost efficiency LHCb is interested in buying middle-class data center CPUs. However, as the stack currently only runs on Intel, the price is expected to be rather high. Therefore, LHCb is exploring new architectures. The first one selected is ARM. ARM currently pushes into the data center and HPC market and offers middle-class CPUs for a reasonable price. IBM was not selected as first architecture to be ported, as their CPU pricing is rather high.

5.1. Compiling the Stack with LCG 91

At the start of the compilation, the fastest ARM machine available is the ARMv8 Testsystem and the newest LCG is LCG version 91. At first LCG has to be compiled, afterwards in order the projects Gaudi, LHCb, Lbcom, Rec and Brunel. To build the stack `cmake` 3.9 is used and the compiler is Gcc 6.2. At this point it is not known that the ARMv8 Testsystem never throws floating point exceptions (FPE) - not even when directly requested by the software. This is found out when switching to the ThunderX2 and analyzing the problems occurring there.

5.1.1. LCG

The compilation of LCG needs multiple modifications. Therefore the initial goal is the compilation of the sub-project ROOT. The following dependencies have to be excluded in the build: `R`, `rpy2` and `cx_oracle`. Aside from this, multiple missing libraries have to be installed on the system prior to the LCG compilation (see Appendix A.1). If not all the missing libraries are install in the beginning, some LCG packages can fail as optional libraries for the installation might be missing at a later point. This is especially true for Python.

LCG as standalone build has three different directories. The dependency descriptions with their versions pulled from Gitlab is in the directory `lcgcmake`. `lcgcmake-build` is the directory for the sources and also where the cmake call is done. And `lcgcmake-install` is the directory, where all the finished installations are put. Aside from this, lcgcmake needs LbScripts and CMT for a successful run.

It is to note that a multi-threaded build is most likely to fail, as cmake is not able to resolve all required dependencies in the correct order. This is because not all dependencies are declared within the dependency files. ROOT is missing files in lcgcmake-install. The missing sub-directory `etc/` can be copied from lcgcmake-build.

5.1.2. Gaudi

Gaudi requires multiple changes. In `cmake/GaudiBuildFlags.cmake` the flag

`-Wl,-z,max-page-size=0x1000`

has to be replaced by

`-Wl,-z,common-page-size=0x1000`

else the build fails when trying to link the libraries. For the runtime selection of the specific Vcl intrinsic set, the instruction level has to be set to 0. This selects the generic, non-vectorized version without even checking which intrinsic sets are available. The intrinsic set checking uses only Intel-compatible assembler commands, which would crash the program on a different platform.

As already stated in Section 5.1.1, the missing ROOT directory has to be copied into lcgcmake-install. Furthermore, the Python script `GaudiPolicy/scripts/CTestXML2HTML` uses the information in `/proc/cpuinfo`. However, the identifiers used in `cpuinfo` differ depending on the platform. For example, on x86-architecture the identifier for the tupel containing all the available flags of the machine is called `flags` whereas on the aarch64 machine it is called `Features`. Furthermore, cmake compresses the stdout to save disk space. To have a human-readable summary of the tests, the compressed stdout has to be decoded again. A patch for this is available, which is cherry-picked into the current branch.

The last change required for Gaudi is to force the ARM compiler to expand `char` automatically to the same signedness as on Intel architectures. For this `-fsigned-char` has to be added to `cmake/GaudiBuildFlags.cmake`. More about this in Section 5.1.3

5.1.3. LHCb

LHCb is the first project requiring changes specifically because of vectorization.

Changes for Vectorization

Before LHCb, Vcl only had some preliminary support for floating-point operations on aarch64 platforms due to its use in `cross_kalman`. Now, it has to be extended for integer vectorization. This build only supports statically-linked Vcl use. However, LHCb also uses dynamically-linked vectorization. There, all different vectorization sets are compiled and during runtime, depending on the architecture, dynamically selected. The dynamic selection is only available for Intel intrinsics and therefore has to be deactivated for aarch64. Instead, a generic non-vectorized version is used.

The integer expansion of Vcl has a problem: Intel does not distinguish between an unsigned and signed integer vector. The integer data type for 128-bit register is just called `__m128i`. Even though VCL offers an unsigned integer vector, it is not really one. Some of the internal intrinsics used are not the one for unsigned, but for signed integer vectors. For example, there is only a function to convert a float vector to a signed integer vector. But no conversion from float to unsigned integer as ultimately, internally there is no difference between signed and unsigned. However, the NEON intrinsics allow to specify more clearly which kind of data type is used. They have both versions, unsigned and signed integer. As a result a conversion from float to signed integers differs from the conversion of float to unsigned integer. Therefore I

experimentally added the function `truncate_to_uint` to Vcl. But as it breaks the homogeneity of the interface, the better solution is it to use only signed integer vectors.

Aside from this, the files enforcing specific Vcl implementation for Intel have to be removed from the build process specified in `Kernel/LHCbMath/CMakeLists.txt`. Tests for those files are also removed. They are all guarded by checking on which architecture the build process runs.

Signedness of Char

The default signedness of the primitive data type `char` depends on the platform. For the Intel platform the default `char` expands to `signed char`, whereas on the ARM architecture the default `char` expands to `unsigned char`. This different default behavior resulted e. g. in a miss-calculation of a one-at-a-time hash function. To solve this problem one can either directly specify which signedness of `char` should be used, or Gcc offers to use `-fsigned-char` forcing on ARM machines to change the default expansion of `char` to be signed.

Other Changes

In `Kernel/LHCbKernel/Kernel/FPEGuard.h` the assembler command `fwait` has to be removed. The commit [10] suggests that this can be done without any problems. Additionally, the platform information for the aarch64 machine has to be added to `PlatformInfo.cpp` in `Kernel/LHCbKernel/`.

In `Tools/XmlTools/src/component/EntityResolverDispatcher.cpp` there is a problem with the lambda used with `declareUpdateHandler`. Moving the lambda expression into a new method solved the problem.

In `Rich/RichUtils/RichUtils/RichVectorRayTracingUtils.h` the vectorized functions have to be duplicated and adjusted for Vcl. This also includes extending the templates to guide the compiler to find the right function for Vc and Vcl. An example of this is shown in Code Section 5.1. For this, the template deciding between vectorized and scalar version is extended by another type check. If the type is a vector then it additionally checks if it is a Vcl data type or not. Depending on this it selects the proper implementation - either scalar, Vc or Vcl.

Additionally, after switching to a different machine, the change in Section 5.2.2 is required.

```
// Old, only Vc use expected
template < typename POINT, typename VECTOR, typename FTYPE,
  typename = typename std::enable_if<
              !std::is_arithmetic<typename POINT::Scalar>::value &&
              !std::is_arithmetic<typename VECTOR::Scalar>::value &&
              !std::is_arithmetic<FTYPE>::value >::type >
inline typename FTYPE::mask_type intersectSpherical ( const POINT&
   position,
              const VECTOR& direction,
              const POINT& CoC,
              const FTYPE radius,
              POINT& intersection )
{ /* method body */ }
```

```
// New for Vc
template < typename POINT, typename VECTOR, typename FTYPE,
  typename = typename std::enable_if<
              !std::is_arithmetic<typename POINT::Scalar>::value &&
              !std::is_arithmetic<typename VECTOR::Scalar>::value &&
              !std::is_arithmetic<FTYPE>::value &&
              !std::is_same<FTYPE, Vec2d>::value >::type >
inline typename FTYPE::mask_type intersectSpherical ( const POINT&
    position,
                    const VECTOR& direction,
                    const POINT& CoC,
                    const FTYPE radius,
                    POINT& intersection )
{ /* method body */ }

//New for Vcl
template < typename POINT, typename VECTOR, typename FTYPE,
  typename = typename std::enable_if<
              !std::is_arithmetic<typename POINT::Scalar>::value &&
              !std::is_arithmetic<typename VECTOR::Scalar>::value &&
              !std::is_arithmetic<FTYPE>::value &&
               std::is_same<FTYPE, Vec2d>::value >::type >
inline std::bitset<2> intersectSpherical ( const POINT& position,
                    const VECTOR& direction,
                    const POINT& CoC,
                    const FTYPE radius,
                    POINT& intersection )
{ /* method body */ }
```

Code Section 5.1 Changes in the template to distinguish between Vc and Vcl

5.1.4. Lbcom

Lbcom requires solely one change. In `Rich/RichFutureTools/src/RichRayTracing.cpp` is Vc used as vectorization library. As shown in Table 4.2, Vc is currently in progress to have NEON intrinsic support. However, there is no working version available. Also, Vc has no working scalar implementation. Asking the maintainer, he replied, scalar emulation would be possible but there are some guarding `#ifdef's` missing. The fastest solution is a hack to use Vcl instead of Vc.

First step for using Vcl is to disable the dynamic selection of different Intel intrinsic sets. Instead, the generic one is used, which will be changed to use Vcl. The only file using Vc is `Rich/RichFutureTools/src/RichRayTracing.icpp` which is embedded in the different dynamic versions of `Rich/RichFutureTools/src/RichRayTracing.cpp`. This is possible as in Vc the developer does not have to decide which vector width is used, instead Vc can automatically select the right vector width depending on the underlying available intrinsic sets. For the aarch64 port, the vector width is fixed to two doubles, which fills up the 128-bit register width.

At the beginning of `RichRayTracing.icpp`, typedefs are declared to have a more readable version of data types. These are overwritten by versions using Vcl instead of Vc. For example,

instead of `using VcFloat = Vc::Vector<ScFloat>`, `VcFloat` now is a synonym for the Vcl data type `Vec2d`. More difficult is the replacement of `VcAllocVector`. It is a wrapper of a `std::vector` of Vc data types, with the needed alignment. For aarch64 an alignment of 16 bits is required, thus `VcAllocVector` is replaced by `alignas(16) std::vector<VcPoint>`. Those and more changes are shown in Code Section 5.2.

Furthermore, the templates of `LHCb/Rich` and ROOT functions being used have to be extended to cope with Vcl as vectorization back end. For the changes in LHCb see Code Section 5.1 and for the change in ROOT see Table 5.3.

```
//old: vector type
using VcFloat   = Vc::Vector<ScFloat>;
//new
using VcFloat   = Vec2d;

//old: vector size
if ( VcFloat::Size-1 == ivc )
//new
const  int VcFloatSize = sizeof(VcFloat) / sizeof(ScFloat);
...
if ( VcFloatSize-1 == ivc )


//old: vector of Vc vector type
VcAllocVector<VcPoint> points( NVC, VcPoint(startPoint.x(),startPoint.y(),
   startPoint.z()) );
//new
alignas(16) std::vector<VcPoint> points( NVC, VcPoint(startPoint.x(),
   startPoint.y(),startPoint.z()) );

//old: insert into vector
x[ivc] = dir.x();
//new
x.insert(ivc, dir.x());


//old: check mask
if ( any_of(mask) )
//new
if ( mask.any() )
```

Code Section 5.2 RichRayTracing: Changes to switch from Vc to Vcl

```
// only this file needs to be extended
// file name: lcgcmake-install/ROOT/6.10.06/aarch64-centos7-gcc62-opt/
   include/Math/GenVector/Plane3D.h

//add
#include "VectorClass/vectorclass.h"

//change template
template <typename SCALAR = T, typename std::enable_if<!std::is_arithmetic
   <SCALAR>::value && !std::is_same<SCALAR,
   Vec2d>::value>::type * = nullptr >
void Normalize()
```

```
{
  // normalize the plane
  SCALAR s = sqrt(fA * fA + fB * fB + fC * fC);
  // what to do if s = 0 ?
  const auto m = (s == SCALAR(0));
  // set zero entries to 1 in the vector to avoid /0 later on
  s(m)           = SCALAR(1);
  fD(m)          = SCALAR(0);
  const SCALAR w = SCALAR(1) / s;
  fA *= w;
  fB *= w;
  fC *= w;
  fD *= w;
}

// add Vcl version of the same function
template <typename SCALAR = T, typename std::enable_if<!std::is_arithmetic
    <SCALAR>::value && std::is_same<SCALAR, Vec2d>::value>::type * =
    nullptr >
  void Normalize()
  {
    // normalize the plane
    SCALAR s = sqrt(fA * fA + fB * fB + fC * fC);
    // what to do if s = 0 ?
    const auto m = (s == SCALAR(0));
    // set zero entries to 1 in the vector to avoid /0 later on
    s = select(m, 1.0, s);
    fD = select(m, 0.0, fD);

    const SCALAR w = SCALAR(1.0) / s;
    fA *= w;
    fB *= w;
    fC *= w;
    fD *= w;
  }
```

Code Section 5.3 ROOT: Changes required for Vcl being used in RichRayTracing

5.1.5. Rec

The project Rec requires two changes, both based on the use of vectorization. One change is in the file `Rich/RichRecUtils/RichRecUtils/QuarticSolverNewton.h`. Here, a vector of size four is used. Depending on a macro variable it is either float or double. But as the register width of the aarch64 platform is 128 bits, a double vector of size four is not possible, and thus an unknown type in Vcl. To solve this problem, the vector type is fixed to float.

The other change is in `Pr/PrPixel/src/PrPixelTrack.cpp`. There, two functions are written in raw Intel SSEx intrinsics. They are replaced by Vcl to be architecture-independent. Both functions use a lot of shuffle commands. Replacing those is prone to errors. In Code Section 5.4 examples are shown how to replace the code. Internally, the blend and permute commands are replaced by a table look-up intrinsic in NEON, as there are no direct shuffle or permute commands. This might be due to the fact, that the ARM processor is a RISC architecture

(Reduced Instruction Set Computer) - having less variety in hardware implemented commands compared to Intel's CISC (Complex Instruction Set Computer) processors.

One wrongly used replacement of an intrinsic resulted further away in a division by zero FPE (see Section 5.2.2).

```
v1 = _mm_shuffle_ps(v2, v1, _MM_SHUFFLE(3,2,1,0));
//is replaced by (_MM_SHUFFLE is read: right to left, and the first two
   numbers refer to v1, the last two numbers refer to v2 - each with
   index 0-3)
v1 = blend4f<0, 1, 6, 7>(v2, v1);

v3 = _mm_shuffle_ps(v2, v2, _MM_SHUFFLE(3,2,3,2));
// is replaced by
v3 = permute4f<2, 3, 2, 3>(v2);
```

Code Section 5.4 Replacement of SEEx shuffle intrinsics by Vcl

5.1.6. Brunel

Nothing is changed in Brunel, but the state is frozen when it was pulled from Gitlab.

5.2. Compiling the Stack with LCG 92

After the successful build and validation of the entire stack with LCG 91, the commercial ARM platform Cavium ThunderX2 was selected for performance testing. At that time, the new version 92 of LCG was available. Therefore it was decided, to not only switch the ARM machine, but also to integrate all required changes for LCG 92.

5.2.1. LCG 92

With the experience of LCG 91, the installation of LCG 92 was faster. As on the old machine before, many missing dependencies had to be installed on the machine before the actual installation process could be started - though it was a slightly smaller subset of the list shown in Appendix A.1.

For LCG 92 some of the package versions had to be changed. For this LCG 91 was a guideline, though it was tried to keep the current LCG 92 versions as much as possible. Also, instead of commenting out unsupported packages, they were wrapped with if-statements checking for the platform. This allows to use the code on different platforms without breaking it.

5.2.2. Problems

Switching to the Cavium ThunderX2 resulted in unforseen, platform-related problems. It had not been found before, as - unkown to everyone - the ARMv8 Testsystem ignores all floating point exception.

Cast from double to unsigned int

The problematic code lines in the file `Det/STDet/src/Lib/DeSTSensor.cpp` of the LHCb project are shown in Code Section 5.5. The function `floor` returns a double value rounded towards negative infinity. After this, the double value is cast into an unsigned integer. In some cases `floor` returns a negative value. On the Cavium ThunderX2, this raises the floating point exception FE_INVALID.

```
1  unsigned int DeSTSensor::localUToStrip(const double u) const{

   // convert local u to a strip
   unsigned int strip;
5  if (m_xInverted == true){
     strip = (unsigned int) floor(((m_uMaxLocal-u)/m_pitch) + 0.5);
   }
   else {
9    strip = (unsigned int) floor(((u-m_uMinLocal)/m_pitch) + 0.5);
   }

   return (isStrip(strip) ? strip : 0);
13 }
```

Code Section 5.5 DeSTSensor.cpp

The ARM User Guide defines that one of the cases FE_INVALID is raised is when "Converting a floating-point number to an integer if the result does not fit"[28]. However, both ARM machines show different behaviors. To understand this, the test program shown in Code Section 5.7 is used on different machines and with different compilers. This test program tries to cast different large double values to unsigned int. It is run two times, once with the floating-point exception enabled and once with the floating-point exception disabled. The result is shown in Table 5.1.

Looking into the disassembly shows that Intel uses a cast from `double` to `signed int` and then reinterprets the result to `unsigned int` - whereas ARM has a specific intrinsic to cast from `double` to `unsigned int`.

In order to obtain a fast solution for the problem, the behavior of Intel has to be recreated. This means casting from `double` to `signed int`, and then casting to `unsigned int`, as shown in Code Section 5.6.

```
   unsigned int DeSTSensor::localUToStrip(const double u) const{

3  // convert local u to a strip
   unsigned int strip;
   if (m_xInverted == true){
   // double -> int -> unsigned int necessary on ARM platform, else
     floating point exception
7  //  if trying to convert negative doubles to unsigned int
     strip = static_cast<unsigned int>(static_cast<int>(floor(((m_uMaxLocal
     -u)/m_pitch) + 0.5)));
   }
   else {
11 // double -> int -> unsigned int necessary on ARM platform, else
     floating point exception
```

```
  //  if trying to convert negative doubles to unsigned int
    strip = static_cast<unsigned int>(static_cast<int>(floor(((u-
   m_uMinLocal)/m_pitch) + 0.5)));
  }

  return (isStrip(strip) ? strip : 0);
}
```

Code Section 5.6 Solution: DeSTSensor.cpp

```
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif

//might be unnecessary
#define __USE_GNU
#pragma STDC FENV_ACCESS ON

#include <fenv.h>
#include <stdio.h>

int main() {
  feenableexcept(FE_INVALID);
  int desc = 0;

  if (desc = 0) {
    volatile double f = -1.;
    unsigned int x = (unsigned int)f;
    printf("x %u\n", x);
  } else {
    volatile double h = 65000;
    volatile double f = 1.e10;
    volatile double g = 1.e100;

    unsigned int u = (unsigned int)h;
    printf("u %u\n", u);
    unsigned int x = (unsigned int)f;
    printf("x %u\n", x);
    unsigned int c = (unsigned int)g;
    printf("c %u\n", c);
  }

  return 0;

}
```

Code Section 5.7 Test program for FE_INVALID

Table 5.1. Results of the test programm FE_INVALID

FE_INVALID enabled					
	Build type	Gcc 7.2			
Double value		-1.0	65000	10^{10}	10^{100}
Testsystem (ARMv8)	native	0	65000	Max_Int	Max_Int
ThunderX2 (ARMv8)	native	Exception	65000	Exception	Exception
E5-2630 v4 (Intel)	Gcc7.1, native	Max_Int	65000	1410065408	Exception
FE_INVALID disabled					
	Build type	Gcc 7.2			
Double value		-1.0	65000	10^{10}	10^{100}
Testsystem (ARMv8)	native	0	65000	Max_Int	Max_Int
ThunderX2 (ARMv8)	native	0	65000	Max_Int	Max_Int
E5-2630 v4 (Intel)	Gcc7.1, native	Max_Int	65000	1410065408	0
Further Information					
Max_Int = 4294967295 10^{100} mod Max_Int = 1410065408 Tested with Gcc 6.2, 7.2 and 4.8 (if available). Some were natively built on the machines, some not. Nevertheless, the results were always consistent.					

Division by Zero Exception

After fixing the previous floating-point exception, a new floating-point exception occurred. This time, it is a division by zero being thrown while computing a covariance. Finding the source of the problem is tedious. First, like with the FPE before, the behavior of division by zero FPEs on the different architectures is analyzed. In Table 5.2 it is shown that the Intel platform and the ThunderX2 exhibit the same behavior for floating point division by zero. Therefore, without LCG the LHCb stack is transferred to an Intel machine and all changes done for the ARM platform are reversed. The Brunel test is run and shows no errors. So step by step, starting with LHCb, followed by Lbcom and Rec, the changes for ARM are reapplied. Each time the Brunel tests are run to see which change results in the FPE. In the end, the changes in Rec creates the FPE. One of the SSEx intrinsics was wrongly replaced, and wrote only zeros into a vector instead of proper values.

Applying the changes to the ThunderX2 and running the Brunel tests shows no further errors being thrown.

Table 5.2. Results of the test programm FE_DIVBYZERO

FE_DIVBYZERO enabled				
	Build type	Gcc 7.2		
Data type		Vector<2>	Double	Integer
Testsystem (ARMv8)	native	<Infinity, Infinity>	Infinity	0
ThunderX2 (ARMv8)	native	Exception	Exception	0
E5-2630 v4 (Intel)	Gcc7.1, native	Exception	Exception	Exception
FE_DIVBYZERO disabled				
	Build type	Gcc 7.2		
Data type		Vector<2>	Double	Integer
Testsystem (ARMv8)	native	<Infinity, Infinity>	Infinity	0
ThunderX2 (ARMv8)	native	<Infinity, Infinity>	Infinity	0
E5-2630 v4 (Intel)	Gcc7.1, native	<Infinity, Infinity>	Infinity	Exception
Further Information				
Tested with Gcc 6.2, 7.2 and 4.8 (if available). Some were natively built on the machines, some not. Nevertheless, the results were always consistent.				

5.3. Validation

The validation of the port to the aarch64 platform is done by evaluating the results of the Brunel tests. For this, three different Brunel tests are done

- ARM, aarch64, CentOS 7, Gcc 6.2
- ARM, aarch64, CentOS 7, Gcc 7.2
- Intel, AVX2, CentOS 7, Gcc 6.2, no ARM changes applied

The Brunel test on Intel is done to evaluate the difference between the reference files included in Brunel and the actual results of the current LHCb stack version before the ARM changes. Here it is shown, that some tests fail as files are not found. Otherwise, there are no numerical differences compared to the Brunel test reference files. The results of the Brunel tests on the ARM platform are slightly different depending on the compiler being used. This is expected behavior.

Marco Cattaneo is one of the persons at LHCb experienced in what are acceptable deltas compared to the reference files. He evaluated the Brunel tests on the ThunderX2, Gcc 7.2 and concluded that the numerical differences are small enough. However, most numerical differences occur in the RICH section. Therefore he suggested looking again at the change in the `QuarticSolverNewton.h` (see Section 5.1.5), if it is possible to change back to `double`. However, as the computations require a vector of size four, the change to double on aarch64

is only trivially possible by switching to scalar computation. Running the tests again shows some numerical difference. However, they are so few and small that they cannot account for the differences being observed by switching the computer architecture.

5.4. Performance

Measuring the performance is done on different machines with different settings. The limitations are given by the LHCb stack. As it is an old version out of October 2017, there is no multi-threading possible. Instead multiple processes are spawned. It was considered to use a forking mechanism to reduce the needed memory, but the code for it uses assembler instructions. And as the current version of the LHCb stack already uses multi-threading, it is not worth the time porting the assembler instructions to aarch64. Furthermore, switching to a new version of the LHCb stack is not possible, as in the current version Vc is used in a larger quantity - and which still does not have aarch64 support.

The performance is measured on the machines listed in Table 5.3 and the analysis is run with following setup

- Spawning multiple processes
- Spawning multiple processes with numactl
- Spawning multiple processes with hwloc-bind

Table 5.3. Machines for the performance test

	E5-2630 v4	ThunderX2
Architecture	Intel	ARM
Platform	x86_64	aarch64
CPUs	40	224
Threads per core	2	4
Cores per socket	10	28
Sockets	2	1
NUMA nodes	2	2
RAM (GB)	64	255
Largest intrinsic set	AVX2	NEON

5.4.1. Problems

File Descriptors

When running more than 100 processes on the ThunderX2 some processes randomly failed, due to running out of the file descriptors. File descriptors handle the interaction with input and output resources. Often, one resource has multiple file descriptors. Each machine has a user and system-wide limit of how many file descriptors are supported. On the ThunderX2 the normal user limit of file descriptors is 1,024. For the performance tests, the limit is increased to 65,000. Aside from the system-wide file descriptor limit, the limit of the network drive `/cvmfs`

which is used by Brunel, is increased to 265,000. To prevent similar problems occurring on the E5-2630 v4 the limit is also increased there.

numactl Behavior on ThunderX2

The ThunderX2 being used is an engineering sample. It has four different nodes with different operating systems. First, a node was used with Red Hat Enterprise Linux (RHEL) as operating system. However, as shown in Figure 5.1 when doing the performance tests using numactl, the performance deteriorates greatly. To understand this problem the topology of the NUMA nodes, the CPUs and their caches is analyzed. Unfortunately, the firmware of the RHEL is not able to show the topology correctly, e. g. checking the number of CPUs, it correctly states 224 cores, but wrongly states it has 68 real cores and three hyper threads per core. The correct number would be 224 cores, of which 28 are real cores with four hyper threads per core. To analyze the problem further, the performance test is also done on a CentOS node of the ThunderX2, which recognizes the topology correctly.

Doing the performance tests on the CentOS node showed the same behavior as on the RHEL node. Again, using numactl results in a significant drop in the performance. After further discussions, someone suggested to use `hwloc-bind` instead of numactl. Like numactl, hwloc-bind allows to bind processes to NUMA sockets and CPUs. The achieved performance by using hwloc-bind corresponds to the expected behavior of having a better performance when binding processes to NUMA nodes. To check, if the results depend on either using numactl or hwloc-bind, hwloc-bind is also run on the RHEL node. Here, even though complaining about some wrong intersection within the topology, hwloc-bind performs as expected - having a better performance compared to no NUMA-binding.

To understand the numactl behavior further tests are done. For this a run is done on the Intel machine binding each process to one CPU in the following order: CPU1#realCore, CPU1#hyperthread, CPU2#realCore, CPU2#hyperthread, ... Binding the processes in such a manner removes the advantage of having multiple cores and hyper threads. Each core is fully loaded before another core is used. The resulting performance starts on a lower level compared to the non-binding version and increases linearly. In the end, on a fully loaded machine it performs the same compared to the other versions. Looking at the ThunderX2, the measured behavior of numactl also increases linear, showing the same symptoms as the 'process locked to one CPU'. Therefore it is probable that the numactl binding order used on Intel is not compatible with the ThunderX2. No further tests to understand the numactl binding are done, as hwloc-bind is giving the required results for the evaluation.

5.4.2. Results

The results of the performance test are shown in Figure 5.1 and 5.2. Here the blue and green lines belong to the ThunderX2 and the red and yellow lines to the Intel E5-2630. The legend includes all information about the specific run - which machine and operating system, how many events per process are run, if and which kind of NUMA binding is used. A NUMA binding containing 'node' states that the processes are bound to a NUMA socket. For numactl this includes also binding the process to all the CPUs belonging to the specific socket, whereas for hwloc-bind only the socket is defined. A NUMA binding containing 'cpu' describes a run where each process is bound to a single cpu - incrementally fully loading one core with work before loading the next core.

In the figures the scaling behavior shows the difference between using real cores and using hyper-threading. For the E5-2630 there are 10 real cores per socket and it has two sockets. So up to 20 processes the machine has a linear scaling behavior and afterwards the performance increase is reduced. Between 21 and 40 processes it has also a linear scaling behavior but with a reduced scaling factor compared to just using the real cores. Having more than 40 processes - which is more than the maximum amount of cores the machine has - the performance stagnates. The additional processes use the not used resources on the machine but at one point the performance decreases because of scheduling and communication overheads. Similar behavior is seen on the ThunderX2. This machine has a total of 56 real cores and four times hyper-threading. The major decrease in the scaling factor is seen when reaching more than 56 processes. The next change in the scaling factor is seen at 112 processes when the third hyper-thread is started. The result of using the fourth hyper-thread is not significantly visible in the graph. The fourth hyper-thread starts at 168 processes. However, it is visible that while using the second hyper-thread there is a linear scaling behavior, but with the third hyper-thread the performance fluctuates - and no pure linear scaling behavior can be observed.

To compare the cost-performance ratio the peak performance of the normalized 'Total events per sec' of Figure 5.1 is taken and divided by the cost of the machine. The resulting bar plot in Figure 5.3 shows that computing the events is significantly cheaper on the Intel E5-2630. The costs for a fully loaded ThunderX2 with using 224 processes for 224 cores is extrapolated as the machine did not have enough RAM to run so many processes. However, there are three things to be considered: First, on the ARM port there is less vectorization as some parts of the vectorization were replaced by a generic scalar implementation during the port. Second, the ported LHCb stack still has to use multiple process instead of being multi-threaded. The increased cache locality introduced by multi-threading has a larger advantage for hyper-threads and the E5-2630 has only two hyper-threads compared to the four hyper-threads of the ThunderX2. Third, the price of the E5-2630 is the price of buying the machine in a bulk, whereas the price of the ThunderX2 is the price for buying a single machine. Therefore, it is expected that the bias towards Intel will be reduced in future cost-performance estimations.

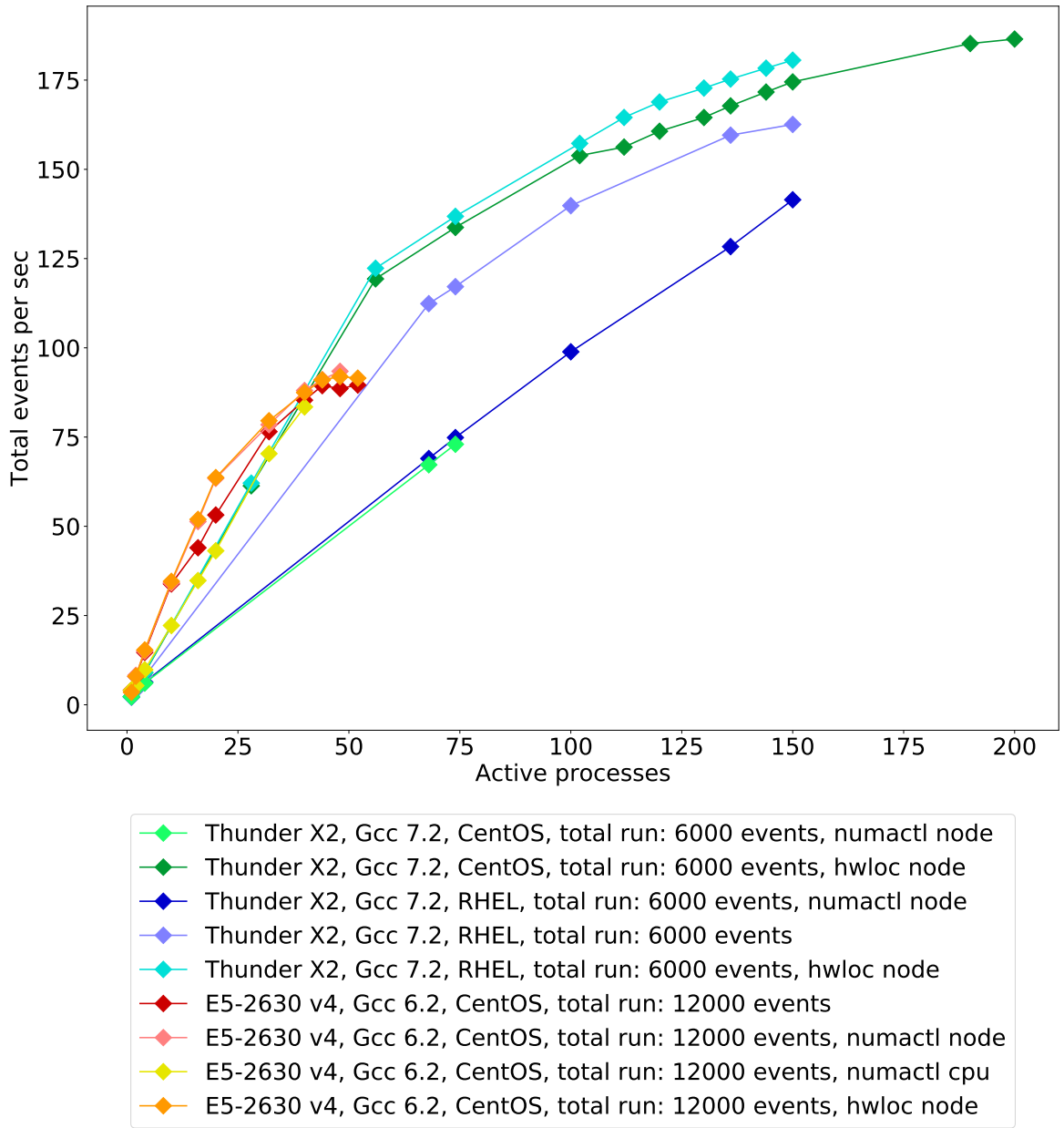


Figure 5.1. Performance based on total time of all processes

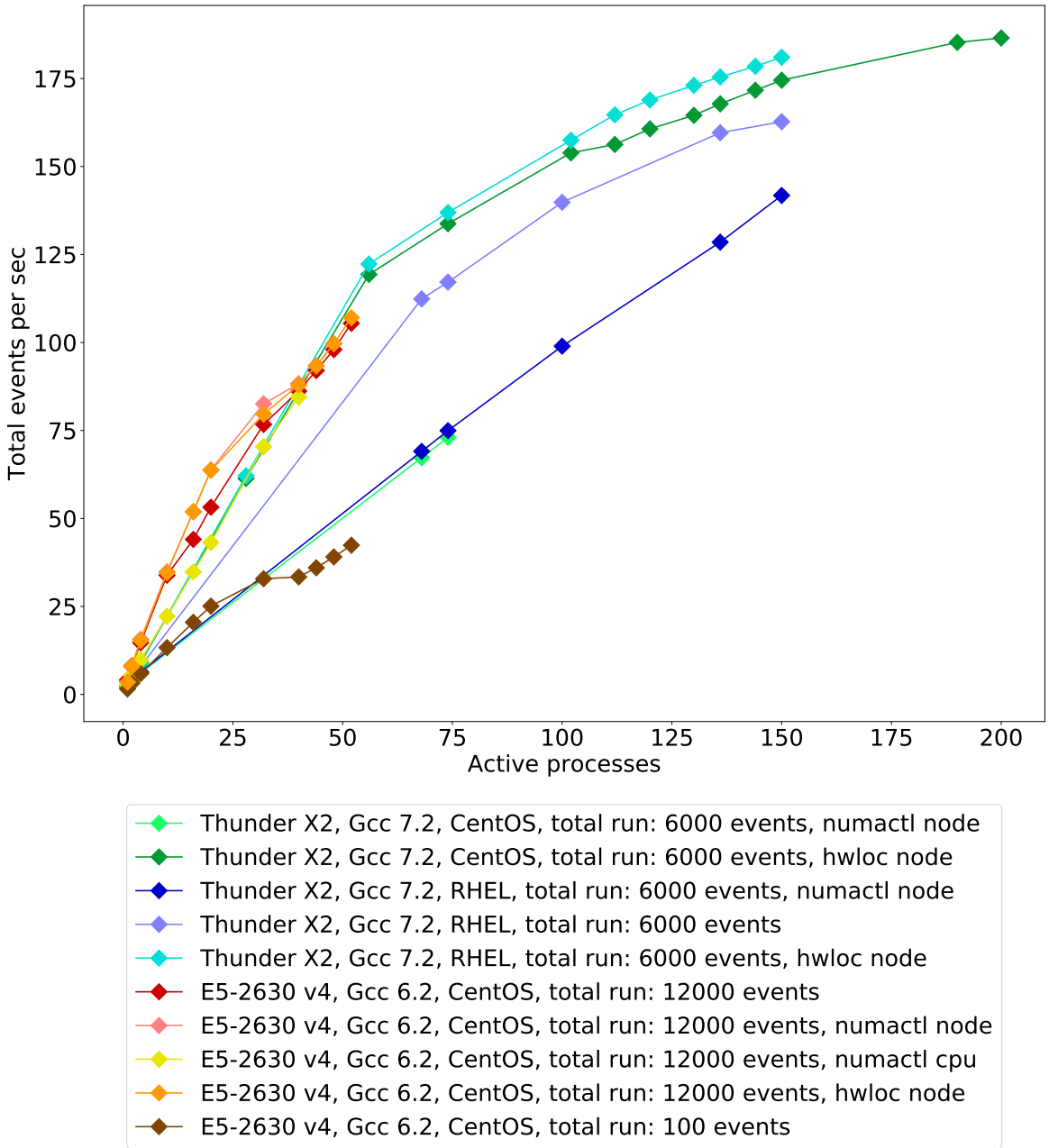


Figure 5.2. Performance based on average time of one event per process

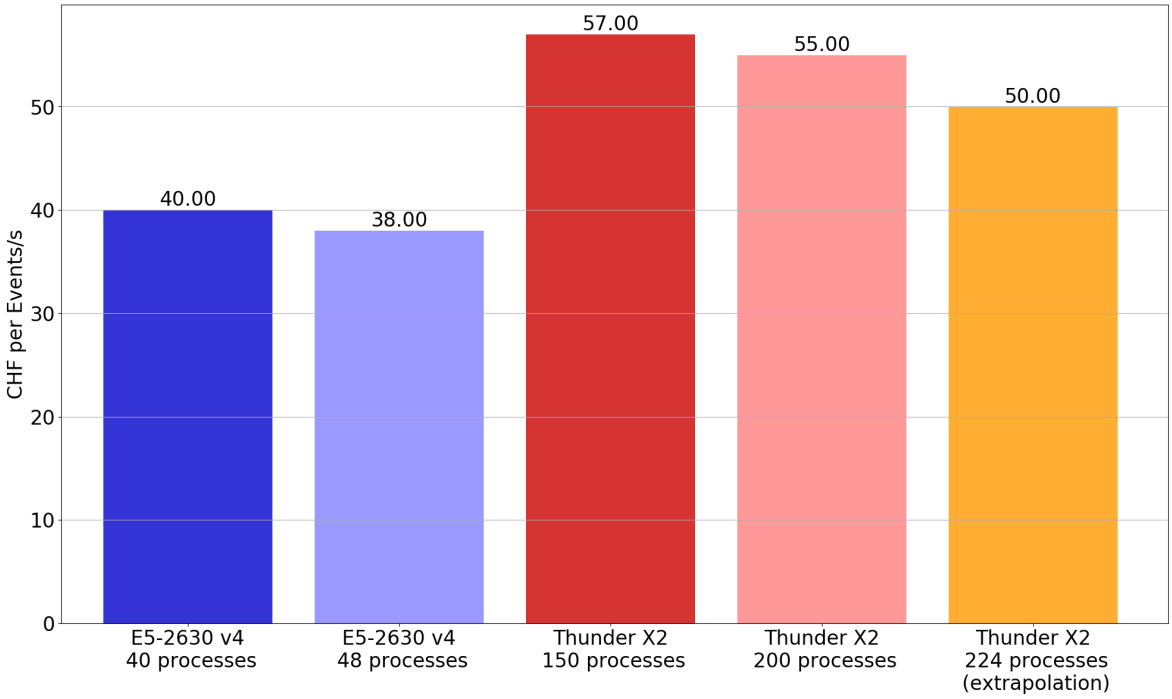


Figure 5.3. Price of the machines per events/s

6. Porting the LHCb Stack to PowerPC

After the successful port of the LHCb stack to aarch64, the port to PowerPC is targeted. As it turns out having the aarch64 port as base and then adding the required changes is the most efficient solution. In view of the time, this master thesis only covers building the stack and its validation.

6.1. Compiling the Stack with LCG 92

For the compilation of LCG 92 at first it is tried to start from the LCG 92 master branch, but too many changes are needed. Instead the required changes are added on top of the aarch64 port. This decision is taken as during working with cross kalman it was noticed that the required changes between aarch64 and ppc64le are rather little compared to the changes needed when based on Intel.

6.1.1. LCG

In the aarch64 version of LCG not supported packages were protected by if statements checking if the platform is aarch64. These guards are either extended by adding pcc64le or are replaced by the inversed statement checking if it is not a Intel x86_64 platform.

Further changes for specific packages are listed in the following section.

CORAL

In CORAL the macro `__linux` has to be replaced by `__linux__` [21]. Further changes are disabling Oracle and certain tests by adding the guards in `CoralTest/qmtest/createconfig.py` and `CoralTest/qmtest/testlist.xml`. Here, again aarch64 can be taken as example.

RELAX root6

For the package RELAX root6 the macro `__linux` also has to be replaced by `__linux__`.

6.1.2. LHCb Stack

Gaudi

As in CORAL, in Gaudi `__linux` has to be replaced by `__linux__`. Additionally, the build flags for Gaudi and the following projects (LHCb, Lbcom, Rec, Brunel) have to be changed. The flag `-march=native`, which automatically activates all proper flags for the current architecture does not exist on PowerPC. Instead `-mcpu=native` has to be used.

LHCb

The major change in LHCb is extending Vcl to support all the functions required by the LHCb stack. This means adding support for integer, unsigned integer, long and unsigned long - and adding permute and blend functions for float. Here, the Vcl behaves differently compared to Intel or aarch64. The Vcl data types have all the `operator()` defined which returns the base intrinsic data type which they represent. However, on PowerPC this is not recognized. Instead a cast to the specific base intrinsic data type is required - which breaks strict-aliasing.

The changes for Vcl also include changing multiple `#ifdef` macros to not only work on aarch64 but also in ppc64le.

Lbcom

In Lbcom a problem occurred because of the Altivec intrinsics. Altivec overloads the defines of bool, pixel and vector. To properly use C++ those macros have to be undefined. This is done in Vcl. However, in Lbcom there are some files declaring a variable named 'pixel'. This breaks the code as it recognizes 'pixel' as data type and not as variable name. To fix it the code in Code Section 6.1 is added at the beginning of the file.

```
1 #ifdef __ALTIVEC__ && defined(pixel)
  #undef pixel
  #endif
```

Code Section 6.1 Undef of the Altivec data type 'pixel' to prevent breaking the code

Rec

Like for the other projects, in some files guards added for aarch64 have to be extended for ppc64le.

Brunel

No changes required.

6.2. Validation

The results of the Brunel test are compared to the aarch64 results. Here, some slight changes can be seen. The overall differences compared to the expected results in Intel are small enough that Marco Cattaneo also validated this port.

7. Summary

This master thesis is just a milestone at the beginning of the increased use of vectorization and the general cross-platform support of the LHCb stack. The exploration of the vectorization libraries has shown that there are potential libraries for LHCb. Vc is of interest due to its use in LHCb and its integration within ROOT. And compared to Vcl, UMESIMD is of interest as it has the scalar emulation and similar performance compared to Vcl. However, it is also shown that there is no perfect solution, yet. The main disadvantage of the libraries excluding Vc is that LHCb would have to maintain them itself. For Vc on the other hand, LHCb has to wait for the missing implementations.

The port of an old version of the LHCb stack to ARM (aarch64) and PowerPC (ppc64le) proved successful. Although, in the cost-performance analysis between Intel E5-2630v4 and the ARM ThunderX2, the ThunderX2 lost (see Figure 5.3) there are several factors which have to be considered: First, the tested LHCb stack is not multi-threaded but has multiple processes. It is likely that a high number of hyper-threads has a larger performance increase for multi-threaded software compared to a software with multiple processes because of the cache locality. Second, there are several places where the vectorized version has been removed and was replaced by generic scalar code - which also results in a performance decrease. And third, the pricing of the machines has to be considered. The price of the Intel machine is the price of being bought in a bulk, whereas the price of the ARM machine is the price of just buying a single one. Therefore it is expected when switching to the multi-threaded, current version of the LHCb stack with full vectorization that the bias towards Intel will be reduced.

8. Outlook

For the future following two actions are planned. One is about the vectorization libraries. As LHCb does not want to maintain a library, it is of interest what future plans big, vectorized projects like ROOT have. ROOT plans for this year to switch from Vc to VecCore [38]. VecCore is another layer on top of vectorization libraries, allowing to switch to different vectorization back ends. It is planned that the scope of VecCore is increased to fully support all vectorized ROOT functions, and to support UMESIMD and Vc as back end. Thus, having UMESIMD as back end would always allow software using VecCore to be platform independent. Therefore, it is very likely that LHCb will also start using VecCore and removing Vc and Vcl from its source code - and maybe even help with implementing their needed features.

Second, for the port of the LHCb stack to PowerPC (ppc64le) the performance analysis is missing. Once this is done, further steps in the platform selection can be taken. However, an additional step might be taken before: the current machine is not a likely candidate for a farm node as it is a machine learning cluster. Therefore it is possible to ask the CERN IT if they have a likely candidate for LHCb and if they have the pricing for it. If yes, the performance tests will be re-run on that machine and a cost-performance analysis is possible.

Afterwards, a big step has to be taken to merge both actions. The long-term goal is making the current LHCb stack cross-platform ready. However, first the big problem of having either a functioning VecCore or having a cross-platform vectorization library which is supported by the LHCb community has to be solved. And to make it feasible, there must be machines available for the automatic continuous integration tests of the stack for ppc64le and aarch64.

A. Appendix

A.1. Missing Dependencies on the ARM Platform

Need to be installed on ARMv8 Testsystem and Thunder X2

```
tbb
tbb-devel

libbsd
libbsd-devel

bzip2-devel
lbzip2
lbzip2-utils

python-virtualenv
python2-bz2file

libtool
libuuid-devel
bison
flex

libxml++
tcl-tclreadline
tcl-tclreadline-devel
tcltls

ncurses-devel
tk-devel
plplot-tk
```

```
// for root
unuruan
root-unuran
ccache
byacc
motif
motif-devel

//for qt5 - opengl stuff
gtkglext-devel.aarch64
gtkglext-libs.aarch64
mesa-libGLw.aarch64

//for scipy
openblas
openblas-devel

// for joblib
readline-devel

doxygen

xz-devel
libxml2-devel
uuid
freeglut
freeglut-devel
tcl-togl
```

Has to be installed on ARMv8 Testsystem, was already installed on Thunder X2

```
gdbm
gdbm-devel
unzip
htop
bzip2
bzip2-libs
patch

libxslt
python-lxml
libxml2
libxml2-python

tcl
tk
```

```
libcurl
openssl
openssl-devel
openssl-libs
pyOpenSSL

ncurses

zlib
zlib-devel

// for root
libXpm-devel
libXmu
libXmu-devel
```

Bibliography

[1] Gaudi User Guide. http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/Gaudi/Gaudi_v9/GUG/GUG.pdf. [Online; accessed January 11, 2018].

[2] Daniel Hugo Cámpora Pérez, Omar Awile, und Cédric Potterat. A high-throughput kalman filter for modern simd architectures. In Dora B. Heras und Luc Bougé, editors, *Euro-Par 2017: Parallel Processing Workshops*, pages 378–389, Cham, 2018. Springer International Publishing. ISBN 978-3-319-75178-8.

[3] Marco Cattaneo. A Project under CMT, 2009. URL http://lhcb-comp.web.cern.ch/lhcb-comp/Support/CMT/project_under_CMT.htm. [Online; accessed January 11, 2018].

[4] CERN. About CERN. Jan 2012. URL http://cds.cern.ch/record/1997225. [Online; accessed January 8, 2018].

[5] CERN. World map of the relationship between CERN and different countries, 2012. URL https://home.cern/sites/home.web.cern.ch/files/image/inline-images/cmenard/carte2017_en.png. [Online; accessed January 8, 2018].

[6] CERN. Member states. Jan 2012. URL http://cds.cern.ch/record/1997223. [Online; accessed January 8, 2018].

[7] CERN. The Large Hadron Collider. Jan 2014. URL https://cds.cern.ch/record/1998498. [Online; accessed January 11, 2018].

[8] CERN. LHC Guide. [Online; accessed January 8, 2018], Mar 2017. URL http://cds.cern.ch/record/2255762.

[9] CERN LHCb. LHCb Detector Geometry. URL http://lhcb-geom.web.cern.ch/lhcb-geom/images/y-LHCb-reoptimized.pdf. [Online; accessed January 10, 2018].

[10] Marco Clemenci. Removed call to 'fwait' in FPE::detail::get (bd931da7) · Commits · Marco Clemencic / LHCb · GitLab. URL https://gitlab.cern.ch/clemenci/LHCb/commit/bd931da7ace82ba5bc45071328173cb0dad4f80a. [Online; accessed January 31, 2018].

[11] Gloria Corti. Information for guides, 2014. URL http://lhcb-public.web.cern.ch/lhcb-public/en/LHCb-outreach/documentation/LHCb-InfoForGuides-2014.pdf. [Online; accessed January 10, 2018].

[12] CERN EP-SFT. LCG releases | EP-SFT. URL https://ep-dep-sft.web.cern.ch/document/lcg-releases. [Online; accessed January 11, 2018].

[13] Ramsey Faragher. Understanding the basis of the kalman filter via a simple and intuitive derivation [lecture notes]. *IEEE Signal processing magazine*, 29(5):128–132, 2012.

[14] Christian Färber. Alterungsstudien am Outer Tracker des LHCb Experiments. Diplomarbeit, Ruprecht-Karls-Universität Heidelberg, 2008.

[15] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

[16] Agner Fog. Software optimization resources. C++ and assembly. Windows, Linux, BSD, Mac OS X. URL `http://www.agner.org/optimize/`. [Online; accessed January 19, 2018].

[17] M. S. Grewal und A. P. Andrews. Applications of Kalman Filtering in Aerospace 1960 to the Present [Historical Perspectives]. *IEEE Control Systems*, 30(3):69–78, June 2010. ISSN 1066-033X. doi: 10.1109/MCS.2010.936465.

[18] Colfax International. Clustering Modes in Knights Landing Processors - Colfax Research. URL `https://colfaxresearch.com/knl-numa/`. [Online; accessed January 24, 2018].

[19] Przemyslaw Karpinski. GitHub - edanor/umesimd: UME::SIMD A library for explicit simd vectorization. URL `https://github.com/edanor/umesimd`. [Online; accessed January 19, 2018].

[20] Matthias Kretz. GitHub - VcDevel/Vc: SIMD Vector Classes for C++. URL `https://github.com/VcDevel/Vc`. [Online; accessed January 19, 2018].

[21] Dimitri John Ledkovon und Ulrich Weigand. Bug #1349907 "gcc, powerpc with C++11 standard does not define __..." : Bugs : gcc-4.9 package : Ubuntu. URL `https://bugs.launchpad.net/ubuntu/+source/gcc-4.9/+bug/1349907`. [Online; accessed March 7, 2018].

[22] CERN LHCb. Graph of the dependencies between projects, . URL `http://lhcb-doxygen.web.cern.ch/lhcb-doxygen/brunel/v53r1/dependencies.svg`. [Online; accessed January 11, 2018].

[23] CERN LHCb. The BRUNEL Project, . URL `http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/brunel/`. [Online; accessed January 11, 2018].

[24] CERN LHCb. The LHCb Project, . URL `http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/lhcb/`. [Online; accessed January 11, 2018].

[25] CERN LHCb. Detector Description, . URL `http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/DetDesc/default.htm`. [Online; accessed January 11, 2018].

[26] CERN LHCb. The LBCOM Project, . URL `http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/lbcom/`. [Online; accessed January 11, 2018].

[27] CERN LHCb. The REC Project, . URL `http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/rec/`. [Online; accessed January 11, 2018].

[28] Arm Ltd. Arm Compiler Arm C and C++ Libraries and Floating-Point Support User Guide Version 6.6 | Exception types recognized by the Arm floating-point environment – Arm Developer. URL `https://developer.arm.com/docs/dui0808/latest/floating-point-support/ieee-754-arithmetic/exception-types-recognized-by-the-arm-floating-point-environment`. [Online; accessed January 30, 2018].

[29] Nakul Manchanda und Karan Anand. Non-uniform memory access (numa). *New York University*, 4, 2010.

[30] Esma Mobs. The CERN accelerator complex. Complexe des accélérateurs du CERN. Jul 2016. URL `https://cds.cern.ch/record/2197559`. [Online; accessed January 8, 2018].

[31] Esma Mobs und Melissa Marie Jacquemod. CERN Quick Facts 2017 (English version). CERN Instantané 2017 (version anglaise). [Online; accessed January 8, 2018], Jul 2017. URL `http://cds.cern.ch/record/2274789`.

[32] Ricardo Nobre, Luiz G. A. Martins, und João M. P. Cardoso. A Graph-based Iterative Compiler Pass Selection and Phase Ordering Approach. *SIGPLAN Not.*, 51(5):21–30, June 2016. ISSN 0362-1340. doi: 10.1145/2980930.2907959. URL `http://doi.acm.org/10.1145/2980930.2907959`.

[33] NumScale. GitHub - NumScale/boost.simd: Portable SIMD computation library. URL `https://github.com/NumScale/boost.simd`. [Online; accessed January 19, 2018].

[34] Mike P. An Intro to MCDRAM (High Bandwidth Memory) on Knights Landing | Intel® Software. URL `https://software.intel.com/en-us/blogs/2016/01/20/an-intro-to-mcdram-high-bandwidth-memory-on-knights-landing`. [Online; accessed January 24, 2018].

[35] DIRAC Project. DIRAC Documentation, . URL `http://dirac.readthedocs.io/en/latest/`. [Online; accessed January 11, 2018].

[36] LHCbDIRAC Project. LHCbDIRAC Documentation, . URL `http://lhcb-dirac.readthedocs.io/en/latest/index.html`. [Online; accessed January 11, 2018].

[37] Laura Promberger. Laura Promberger / vectorclass_extended. URL `https://gitlab.cern.ch/lpromber/vectorclass_extended`. [Online; accessed January 19, 2018].

[38] root project. GitHub - root-project/veccore: SIMD Vectorization Library. URL `https://github.com/root-project/veccore`. [Online; accessed March 8, 2018].

[39] Devin Smith. GitHub - edanor/umesimd: UME::SIMD A library for explicit simd vectorization. URL `https://opensource.org/licenses/BSL-1.0`. [Online; accessed January 19, 2018].

[40] Ganga-dev Team. Ganga: Gaudi/Athena and Grid Alliance. URL `http://ganga.web.cern.ch/ganga/`. [Online; accessed January 11, 2018].