



UNIVERSITY
OF AMSTERDAM



VRIJE
UNIVERSITEIT
AMSTERDAM

Master Thesis

Modelling GPU Parallelism in ROOT RDataFrame Histograms

Author: Jolly Chen

1st supervisor: Prof. Dr. Ir. Ana Lucia Varbanescu
daily supervisor: Dr. Monica Dessole (CERN)
2nd reader: Dr. Jakob Blomer (CERN)

*A thesis submitted in fulfilment of the requirements for
the joint UvA-VU Master of Science degree in Computer Science*

22 July, 2024

Abstract

Context. CERN, housing to the Large Hadron Collider (LHC), generates petabytes of collision event data for particle physicists to analyse. ROOT, the standard framework for high-energy physics (HEP) data analyses, offers a declarative interface through RDataFrame, where histogramming is a core operation.

Goal. We aim to predict the runtime performance benefits of offloading computations to the GPU, using histogramming in RDataFrame as a case study.

Method. We propose an analytical modelling approach to predict the runtime of CPU and GPU implementations, employing a systematic strategy that includes microbenchmarking and code analysis. We validate our models using benchmarked runtimes on an AMD EPYC 7402P CPU and NVIDIA A4000 GPU.

Results. Our findings indicate that the models accurately predict trends and performance rankings, although the CPU model tends to overestimate runtime, while the GPU model underestimates it. Consequently, our models are overly optimistic in speedup predictions. Nonetheless, the process of designing and using the model enhances our understanding of the performance bottlenecks.

Conclusions. Designing an accurate performance model for runtime predictions entails a trade-off between accuracy and effort. Despite suboptimal accuracies, the modelling process facilitates a deeper understanding of performance characteristics.

Contents

1	Introduction	1
1.1	Research Question and Approach	2
1.2	Thesis outline	2
2	Background	3
2.1	Runtime Performance Models	3
2.2	Histogramming in HEP	4
2.3	GPU Programming Model	6
3	Methodology	7
3.1	Requirements	7
3.2	Modelling Approach	8
3.3	Experimental Setup	9
3.4	Validation Data	10
4	CPU Implementation	13
4.1	Find Bin	15
4.2	Add Bin Content	28
4.3	Update Statistics	31
4.4	Total Model Validation	34
5	GPU Implementation	38
5.1	Find Bin	40
5.2	Add Bin Content	46
5.3	Update Statistics	53
5.4	Memory Transfers	57
5.5	Total Model Validation	64

CONTENTS

6	Scenario Analyses	66
6.1	Overview of Performance Models	66
6.2	Speedup	69
6.3	Overlapping Computation with Communication	70
7	Related Work	72
8	Conclusion	74
8.1	Summary and Main Findings	74
8.2	Limitations	75
8.3	Future work	75
	References	76
A	Experimental Setup	82
A.1	Test Systems	82
A.2	ROOT Setup	83
B	Microbenchmarking	84
B.1	Measuring Runtimes with Google Benchmark	84
B.2	Microbenchmark Guidelines	86
C	GPU Calibration Microbenchmarks	88

Introduction

CERN hosts the Large Hadron Collider (LHC), the world’s largest particle accelerator, in which millions of protons are collided every second. By studying the properties of the particles that are produced during collisions, physicists aim to get a better understanding of the fundamental structures and forces that constitute our universe. The collisions in the detectors are recorded and stored in a digitised format, simply referred to as an *Events*. Due to the high frequency at which these collisions occur, petabytes of event data are collected that need to be analysed. Processing such a substantial amount of data in a reasonable time requires highly efficient computing solutions.

The most common analysis tool used to analyse event data is ROOT, which is an open-source C++ data analysis framework, specifically designed for High Energy Physics (HEP) use cases [7]. ROOT consists of several components that facilitate efficient analysis, processing, visualisation, and storage of event data. One of these components is RDataFrame [29], which is ROOT’s high-level declarative interface for analysing columnar data, i.e., tabular formats that store the events as rows and the properties for each event in the columns.

Since an event dataset is often too large to fit into memory at once, a typical physics analysis can be roughly described as the process of reading chunks of compressed data, decompressing the data, performing statistical analysis, and accumulating results (i.e., histogramming). RDataFrame can transparently parallelise the operations over the available resources to speed up the user’s analysis. However, while support for multi-threaded [29] and distributed execution [28] already exists, GPU support is not available yet.

In recent years, a notable trend towards increasing heterogeneity has been observed in High Performance Computing (HPC) computing facilities [21]. Instead of having CPU-only nodes, it is now common to find nodes equipped with hardware accelerators such

1. INTRODUCTION

as General Purpose Graphics Processing Units (GPGPUs). This change is motivated by substantial performance gains and an increase in energy efficiency shown in other scientific fields [11, 18]. In this thesis, we aim to determine whether RDataFrame can similarly obtain an increase in performance by offloading computations to a GPU, focusing on one of the core HEP operations – histogramming.

1.1 Research Question and Approach

The runtime performance of RDataFrame histogramming depends on many different parameters, e.g., the implementation version, system specifications, the number of input values, and the histogram size. Benchmarking the application for every combination is impractical or impossible because, for example, the implementation does not exist yet. Thus, in this study, we aim to *model* the runtime performance to *predict* beyond what we can test due to time constraints or the unavailability of the implementation. Concretely, our research question is:

RQ. How can we predict the runtime performance benefits of offloading computations to the GPU?

In this work, we propose to analytically model the runtime performance on the CPU and the GPU, by utilising microbenchmarks and algorithm analyses. Using RDataFrame histogramming as a case study, we demonstrate our modelling approach. For the purposes of this study, we implemented basic GPU support within RDataFrame to offload histogram computations, for which the performance can be modelled. We describe how an analytical model can be built, how it can be used to evaluate the performance, and what insights it can give us on the runtime performance.

1.2 Thesis outline

The remaining chapters are structured as follows: in Chapter 2, we delve into common modelling techniques to further motivate our choice of analytical modelling. We also introduce the RDataFrame interface and GPU programming terminology. Chapter 3 outlines our strategy for designing and validating an analytical model. In Chapters 4 and 5, we describe the execution of our modelling approach for the CPU and GPU implementation, respectively. Chapter 6 presents the insights gained from our models regarding the GPU performance. In Chapter 7, we discuss the relevance of our work in relation to other studies. Finally, in Chapter 8, we summarise our findings, indicate the limitations of our work, and provide suggestions for future work.

2

Background

In this chapter, we elaborate on the concept of performance modelling and existing modelling techniques to further motivate our choice for using an analytical model. In addition, we introduce the `RDataFrame` interface to better understand the challenges we face when offloading computations to the GPU.

2.1 Runtime Performance Models

Runtime performance models predict the execution time of a software application. The primary motivation for modelling is algorithm or device selection based on the anticipated performance. There exist several techniques for modelling runtime performance, which can generally be classified into three categories: machine learning, simulation, and analytical.

Performance models based on machine learning train a model that predicts an algorithm’s runtime as a function of problem-specific instance features [1, 19]. While it has been shown that these models can give highly accurate predictions, training typically requires a substantial amount of runtime data, and gathering this information is time-consuming and not possible if the implementation is non-existent. Additionally, the results are difficult to interpret, as the influence of the different parameters on the runtime remains hidden. For these reasons, this technique is not suitable for our purposes.

With simulation, the model emulates the execution of a program to determine the runtime [17, 37]. Unlike the machine learning approach, this can give a full understanding of a parameter’s influence on a runtime prediction by tracing the simulation steps. However, depending on the level of detail, running a simulation is commonly slow. Again, this makes the method unsuitable for our purpose.

2. BACKGROUND

In the third approach, analytical modelling, we design a mathematical equation that calculates the execution time based on a set of input parameters. The advantage of this is that we do not have to execute the full program to get a prediction of its runtime. This saves time and does not require an implementation. Therefore, we use an analytical approach for predicting the runtime.

In practice, runtime performance models are prevalent in profiling tools such as Intel VTune [20] and NSight Compute [27]. Using performance models, they can give suggestions for the most optimal thread organisation. For heterogeneous systems, performance models are commonly utilised in runtime systems such as StarPU [4], OmpSs [12], and IRIS [23], which can automatically select the most suitable device to schedule a task on based on the performance prediction.

2.2 Histogramming in HEP

RDataFrame [29, 35] is ROOT's high-level data analysis interface for columnar data. Users define their analysis by declaring a sequence of operations that need to be performed on columnar event data stored in a `RDataFrame` object. The dataframe stores each event in a new row and the properties for each event in the columns, where a column can contain non-scalar values. There are two types of operations: *transformations* that modify the dataframe and return a new `RDataFrame`-like object, and *actions* that aggregate data and return a computed result. Examples of transformation operations are defining a new column and filtering rows, while actions are for instance computing a histogram or the mean.

Listing 2.1: Basic RDataFrame workflow

```
1 ROOT::RDataFrame df(dataset)
2 auto df = df.Define("z", "x + y")
3           .Filter("x != 0")
4           .Define("r2", "x*x + y*y");
5 auto h1 = df.Histo1D("z")
6 auto h2 = df.Histo1D("r2")
7 h1.Draw(); h2.Draw();
```

An example of the RDataFrame workflow is illustrated in Listing 2.1. On the first line, the user loads a columnar dataset into a `RDataFrame` object. This is followed by three transformation operations: defining a new column `z` with the sum of the values in columns `x` and `y`, filtering out rows/events where the value in column `x` is zero, and defining a new column `r2` by adding the squared values of columns `x` and `y`. On lines 5 and 6, we have

two calls to the histogram action that fill a 1D histogram with the values in column `z` and `r2` respectively. Finally, in the last line, we plot the resulting histograms.

Figure 2.1 illustrates a well-known example of a histogram produced using ROOT. Histograms are an essential tool in HEP for determining statistical outliers. A histogram divides a range into smaller intervals, referred to as *bins*. The lower and upper values of a bin are called the *edges*. Given input values (*coordinates*) from a column in the dataframe, we determine which bins need to be filled based on which intervals contain the coordinate values. Starting with zeroes, the bin values are incremented by one for each coordinate that ends up in that bin. Alternatively, the increment value *weight* can be specified using the values in another column.

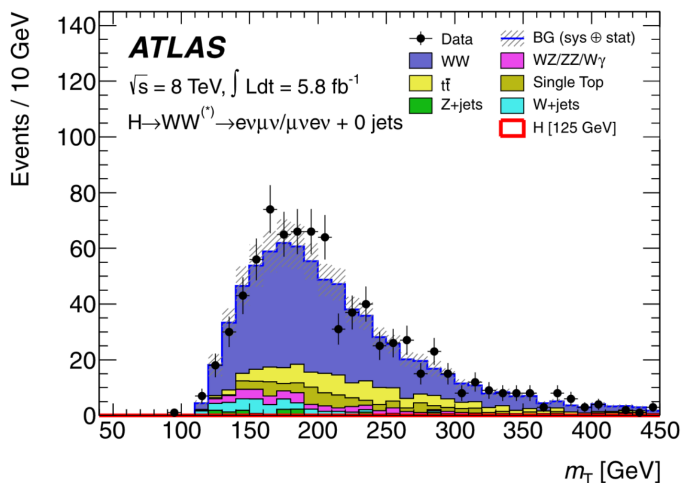


Figure 2.1: Example of histograms produced with ROOT. Taken from the ATLAS collaboration’s original paper on the discovery of the Higgs Boson [39].

A characteristic of HEP analyses is that the input data is typically much larger than what can be stored in the main memory. Hence, instead of loading everything at once, we loop over the event data to load and process the events in manageable chunks. An important aspect of the RDataFrame execution is that the transformations and actions are computed in a single loop over the events. RDataFrame operations are lazy in the sense that the method for booking the operations returns a smart pointer to the result, while the actual data processing is only triggered when the results are accessed for the first time. Once the *event loop* is triggered, each operation is processed in sequence for a *bulk* of events. Internally, RDataFrame constructs a directed computational graph of *transformation* and *action* nodes to keep track of the operations. Consequently, multiple aggregation results can be produced after inspecting all the data only once.

2. BACKGROUND

In the latest ROOT release, 6.30, events are processed one by one in the event loop. However, a refactoring of the loop is currently in development to enable *bulk* processing, which allows for CPU vectorisation and offloading to accelerators [14, 15]. The implementations described in this work are based on that development branch.

2.3 GPU Programming Model

In this section, we explain the basics of GPU programming to better understand how a program is executed on a GPU and support the various design choices we make for our GPU models. Throughout this work, we use CUDA terminology to describe the execution model and memory hierarchy. More details can be found in the CUDA programmer’s guide [26].

Execution Model

To compute on the GPU, we write C++ methods (*kernels*) that are executed by N CUDA threads. These threads are grouped into warps, containing 32 threads, that execute instructions in lockstep (i.e., cannot diverge). The warps are further grouped into *CUDA blocks*, which are mapped to the processing units referred to as *Streaming Multiprocessors (SMs)*. In current NVIDIA generations, the number of threads per block is limited to 1024. When referring to all the threads allocated to a kernel, we use the term *grid*.

Memory Hierarchy

The CUDA threads can access data from multiple data spaces during the execution of a kernel. Most notably:

- *thread-private local memory*: these are registers only visible to the current thread.
- *shared memory*: this is shared between threads within the same CUDA block, and its lifetime aligns with the lifetime of the block. Shares space with the L1 cache and is typically used as a user-managed cache.
- *global memory*: this is shared between all threads, persistent over different kernels, and is generally cached in two different levels.

3

Methodology

In this chapter, we describe our methodology for modelling the execution time of the histogram action in `RDataFrame` (e.g., `Histo1D` in line 6 in Listing 2.1), on both the CPU and the GPU. First, we detail the requirements of the implementation and its model. Subsequently, we discuss the adopted strategy for designing the models. Following this, we outline the hardware and data used to test and validate our models.

3.1 Requirements

To declare a histogram action, the user can use the `Histo1D`, `Histo2D`, `Histo3D`, or `HistND` methods which take up to three input arguments:

- 1). A list defining the *characteristics* of the histogram (e.g., name, number of bins, bin edges, dimension).
- 2). Name of the column in the dataframe that contains the *coordinates* of the bins that need to be filled in the histogram.
- 3). (Optional) name of the column containing the *weights*. If this is not specified, each bin corresponding to a coordinate is filled with a weight of 1.

For the definition of the histogram, the user can either specify the number of bins n within a range $[a, b]$ or manually specify the bin edges. In the first case, ROOT defines a histogram with n bins in the given range with the same bin widths (fixed size). In the second case, the histogram can potentially have variable-sized bins. For multidimensional histograms, the user defines the bins for each axis.

Regarding the input data, each dataframe can have a different total number of events (i.e., rows) which are processed in bulks of user-defined size. Additionally, based on the values within the coordinates column, the data distributions in the resulting histogram can differ.

3. METHODOLOGY

In summary, we have the following parameters for which we model the influence on the total runtime:

- Number of events (number of input values)
- Bulk size
- Data distribution
- Number of histogram bins
- Type of histogram bins: variable or fixed
- Dimensionality

One of the primary design objectives of `RDataFrame` is to offer user-transparent task-based parallelism. This entails that parallel execution on multicore, heterogeneous, and distributed systems can be seamlessly enabled without requiring significant changes to the user code. For example, given a piece of code, the user can enable multi-threaded execution simply by adding a call to `ROOT::EnableImplicitMT()` at the beginning. Given that events are independent, the concurrent processing of `RDataFrame` actions for different events represents an embarrassingly parallel workload. Presently, `RDataFrame` contains support for implicit parallelism in multithreaded [29] and multi-node distributed environments [28], but lacks support for GPU parallelism. Therefore, for this study, we implemented support for GPU execution of the histogram action for different values of the parameter described previously.

3.2 Modelling Approach

To create a performance model for a given application, we execute the following steps:

- 1). **Identify components:** first, we determine which parts of our code base influence the runtime of our application. Generally, given two sections that depend on different parameters, we can split them into two separate components and model them individually.
- 2). **Investigate components:** to model the runtime behaviour of a specific component, we need to investigate which parameters have the strongest influence on the runtime. We do this by studying the algorithm, inspecting the code, and/or microbenchmarking the component.
- 3). **Design a symbolic model:** based on our observations in step 2, we design a mathematical performance model. This step can require some creativity, as there is no clear-cut method for translating the observations into an analytical model.
- 4). **Calibrate model:** some parameters in the model may depend on characteristics of the architecture. Take, for example, a model that estimates the execution time based

on the number of addition operations. Depending on the system’s specifications, the latency of a single addition operation can differ. Consequently, to use a model with architecture-specific parameters, we need to calibrate its parameters.

- 5). **Validate individual components:** to determine the accuracy of a model, we visually compare its predictions against measured execution times of the component with a minimal set of parameters.
- 6). **Validate complete model:** after validating the models of the individual components, we verify the complete model that combines the predictions of the sub-models for each component.

Given major inaccuracies, such as incorrect trends or runtime rankings of different versions, we can decide to re-evaluate and re-validate the model’s design. This can be preceded by additional investigative processes (step 2), to examine the behaviour of the components in more detail. Note that this procedure can also be executed recursively at different granularity for the components. Meaning, a single component can be further broken down into several smaller parts, each going through steps 1-5. The models for the smaller components are then used in the complete model for the encompassing component.

3.3 Experimental Setup

For reproducibility, we describe our environment for running investigational benchmarks and collecting runtime information for model validations. In this work, we use a single node on the DAS-6 cluster [5]. Each compute node on this cluster is equipped with an AMD EPYC 7402P 24-core Processor and 128GB of main memory. For the GPU implementation, we use a node with an NVIDIA RTX A4000 GPU and CUDA driver version 545.23.06. More details on the system’s specifications can be found in Table A.1.

The code for the histogram action with GPU support is available in a fork of ROOT on GitHub[10]. The C++ code is compiled using version 12.2.1 of the GCC compiler, while the files containing CUDA code are compiled using `nvcc` from the CUDA Toolkit version 12.3. All code is compiled with compiler optimisations fully enabled using the flag `-O3`. More details on how to build the ROOT framework are provided in the appendix in section A.2.

To measure runtimes on the CPU, we use `std::chrono::steady_clock`. It is a monotonic clock, which ensures that the clock time does not decrease when the physical time increases. Additionally, the time between clock ticks remains constant. To measure the execution of only GPU activity, we use NVIDIA’s NSight Systems profiler with version 2023.4.1. We run each benchmark 5 times, to obtain the average.

3. METHODOLOGY

3.4 Validation Data

As described in the introduction of this chapter, the execution time of the histogram action is influenced by both the characteristics of the input data and the histogram. For the input data, the parameters are the number of events, bulk size, and the distribution of the data. For the definition of the histogram, we have the number of bins, dimension, and type of bins. To examine a diverse set of combinations, we generate the data used for the validation of the models.

Listing 3.1: Benchmark code fragment. Parameters are highlighted in orange.

```
1 // Load data
2 auto pageSource =
    ROOT::Experimental::Detail::RPageSource::Create("Data", file);
3 auto df = ROOT::RDataFrame(
4     std::make_unique<ROOT::Experimental::RNTupleDS>(
5         std::move(pageSource)), {}, bulkSize);
6
7 // Define histogram
8 if (variable_bins) {
9     std::vector<double> e(nbins + 1);
10    for (int i = 0; i <= nbins; i++)
11        e[i] = i * 1. / nbins;
12    mdl = TH1DModel("h1", "h1", nbins, e);
13 } else {
14     mdl = TH1DModel("h1", "h1", nbins, 0, 1);
15 }
16
17 // Fill histogram
18 auto h = df.Histo1D<double>(mdl, "columnName");
19 auto start = std::steady_clock::now();
20 auto &result = h.GetValue();
21 auto end = std::steady_clock::now();
```

The benchmarking application used to collect validation data for different parameter combinations is shown in Listing 3.1. On the first line, we explicitly initialize ROOT using `ROOT::GetROOT()` to avoid initialization overhead from ROOT's interactive C++ interpreter, `Cling`, during the timing measurements. In addition, we explicitly define the data type of the `Histo1D` call to prevent just-in-time compilation for type inference. Both overheads are constant, so omitting them from the performance comparisons will not affect the validity.

3.4 Validation Data

Parameter	Short name	Values ^{1,3}
Number of events	nEvents	50M, 100M, 500M, 1B
Data type	dType	double
Distribution	distr	<i>Uniform</i> , <i>Normal</i> (0.4, 0.1), <i>Normal</i> (0.7, 0.01) <i>Constant</i> (0.5)
Number of bins	nBins	10, 1K, 100K, 10M
Dimension	dim	1
Type of bins	edges	Fixed, Variable
Bulk size	bulkSize	1, 8, 64, 512, 4096, 32768, 262144

Table 3.1: Validation input parameters.

In Table 3.1, we summarise the values used for the various parameters in the validation of the performance models. The input file is generated in the ROOT **RNTuple** format [36], where we fill the **RNTuple** with a single field/column containing randomly generated double-precision (**double**) values. We test with 50 million to 1 billion events in total; this setup is generic and large enough and keeps the runtime within the maximum allowed job duration on the DAS-6 (15 minutes). For the random generation of the event values, we experiment with different distributions: two extreme cases where all bins are filled with the same probability or only always the middle bin (*Uniform* and *Constant*(0.5)²), and two normal distribution that represent the most common histogram shapes (*Normal*(0.4,0.1) and *Normal*(0.7,0.01))³. To generate the values, we used ROOT’s **TRandom3** pseudo-random number generator. Figure 3.1 illustrates the acquired results when filling, for example, a histogram with 1000 equidistant bins using the selected distributions.

In each experiment, we fill a histogram with bins in the range [0,1]. Given that only the number of bins has an impact on the runtime, we select arbitrary values for the range extremes. Within this range, we vary the number of bins; we experiment with small histograms (10), medium-sized (1K), large (100K), and very large (10M). The maximum number of bins that we can test is limited by the runtime and the available memory on our GPUs. Additionally, we only validate the one-dimensional case with double-precision values. This choice was made to limit the parameter space, but our modelling approach can be easily extended to multi-dimensional histograms with other data types.

To test the case where the bins are potentially variable-sized, we manually generate a vector with equidistant edges instead of variably distanced edges. The size of the bins influences the distribution of which bins are accessed. However, we already investigate different

² δ -distribution denoted as *Constant*(μ), where μ is the only non-zero bin.

³Normal distribution denoted as *Normal*(μ, σ), where μ is the mean and σ the standard deviation.

3. METHODOLOGY

distributions by generating input coordinates with different distributions, so experimenting with variable-sized explicit edges becomes redundant.

To validate the runtime predictions provided by performance models of smaller components, we insert timers around each component to benchmark them separately. However, for validating the runtime estimations of the complete model, we only use the timer shown in Listing 3.1.

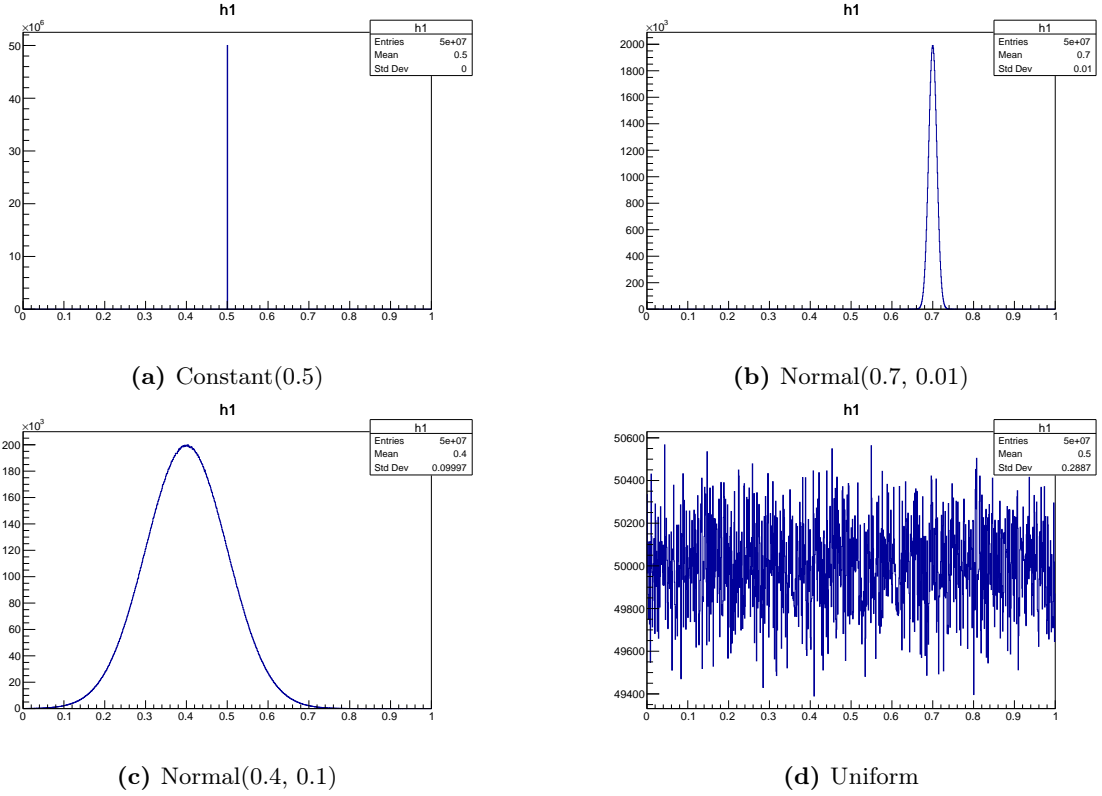


Figure 3.1: Validation data distributions. The figures depict the shaped of the different distributions for a histogram with 1000 equidistant bins with 50000000 events.

4

CPU Implementation

Given a single histogram action within an event loop, we have the process shown in Figure 4.1. When the loop over the events is triggered, we process the events by loading and processing them in bulks and only return the result when all bulks have been processed. Loading event data involves reading a bulk of events from the disk into the main memory, decompressing, and decoding. Once the bulk is retrieved, the data is passed to an action helper class, which processes the histogram action using the relevant histogram class. ROOT version 6 implements several classes for defining a histogram with different data types and dimensionality [34]. To clarify, when the user fills an n -dimensional histogram, they specify n columns containing values that form the *coordinates* of the bins that need to be filled. Optionally, an extra column with the *weights* can be specified to increment the bin with a value other than 1.

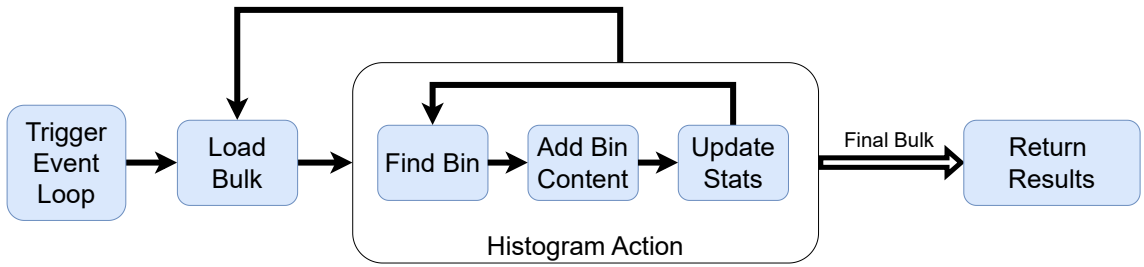


Figure 4.1: Flow diagram of an RDataFrame event loop with one histogram action.

The histogram action can be divided into three different components: 1). *finding* the bin, 2). *adding* the user-specified weight to the bin contents, and 3). *updating histogram statistics*. To not over-complicate the initial model, we only model single-threaded execution, so the three components are processed sequentially for each event within the bulk, and we only process one bulk at a time. Using this model, we can get a lower-bound for multi-threaded execution by simply dividing by the number of threads.

4. CPU IMPLEMENTATION

Since the histogram action consists of a sequence of components that are executed for each event within a bulk, we model the average execution time per event. We assume that the execution time of the histogram action consists of the sum of the execution time of each component for one event, multiplied by the bulk size and the number of bulks (i.e., number of events). The total execution time of the event loop is then comprised of the time spent on loading bulks and processing the histogram action. To summarise, this gives us the model shown in Model 4.1.

Model 4.1: CPU Sequential

$$T_{Total_{CPU}} = \underbrace{\left[\frac{nEvents}{bulkSize} \right]}_{\text{Number of bulks}} * \left(T_{LoadBulk_{CPU}} + bulkSize * (T_{FindBin_{CPU}} + T_{AddBinContent_{CPU}} + T_{UpdateStats_{CPU}}) \right)$$

When moving the histogram action to the GPU, the loading of the bulk into memory remains unmodified, since transfers of data are managed by the CPU on our system. Since this overhead is present in both the CPU and GPU implementation, we assume $T_{LoadBulk_{CPU}}$ to be some constant and only model the histogram action components, without compromising the accuracy of GPU speedups over the CPU predictions. More specifically, we model the following components: $T_{FindBin}$, $T_{AddBinContent}$, and $T_{UpdateStats}$. In the following subsections, we delve into the models of the three components.

4.1 Find Bin

The first component in the histogram action involves finding the bin that needs to be filled. As shown in Algorithm 1, out-of-bounds input coordinates are handled separately. When the user creates a histogram with $nBins$ bins, a flat array is allocated with $nBins + 2$ cells. These cells encompass the user-defined range with two extra cells for capturing underflow or overflow values. Thus, if the input coordinate, $coords_i$, falls outside the histogram’s bin range, either the underflow or overflow bin is selected.

For coordinates within the predefined bounds, the bin calculation varies based on the histogram’s bin type: each bin is of the same size (fixed bins) or some bins are larger than others (variable bins). In the case of an n -dimensional histogram, this process is repeated once per axis, totalling n iterations. Using Horner’s method [8], the coordinates for each dimension are combined into a singular index, facilitating access to the flat histogram array.

Algorithm 1 Pseudocode for “Find Bin”

```

1: procedure HISTOGRAM::FINDBIN( $coords$ ,  $nBins$ ,  $binType$ )
2:    $bin = 0$ 
3:   for  $d$  in  $0, \dots, dim$  do
4:     if  $coords_d < min_d$  then ▷ Underflow
5:        $bin_d = 0$ 
6:     else if  $coords_d \geq max_d$  then ▷ Overflow
7:        $bin_d = nBins_d + 1$ 
8:     else ▷ Variable bins
9:       if  $binType_d == \text{Fixed}$  then
10:         $bin_d = 1 + \left\lfloor nBins_d * \frac{coords_d - min_d}{max_d - min_d} \right\rfloor$ 
11:       else
12:         $bin_d = 1 + \text{BinarySearch}(nBins_d, binEdges_d, coords_d)$ 
13:    $bin = bin * nBins_d + bin_d$ 
   return  $bin$ 

```

Design

The pseudocode illustrates that varying code paths are executed based on the coordinate values, leading to different runtimes. Since the exact number of coordinates falling outside the defined bounds is unknown in advance, we make the assumption that only a negligible number of input values are allocated to the underflow or overflow bins. Consequently, the execution time is primarily influenced by the branch at line 8, which determines the type of histogram bins.

4. CPU IMPLEMENTATION

In the case of fixed bins, the bin calculation is according to a formula, so the amount and type of arithmetic operations do not change based on the input value. One option for modelling this is to count the number of operations and multiply them with microbenchmarked latencies. However, due to compiler optimisations and instruction level parallelism, modelling at such a fine granularity can easily overestimate the runtime, as fewer sequential operations are executed than indicated by the pseudocode. Instead, we microbenchmark the computation of the formula on Line 10 in Algorithm 1 in its entirety. Our model for **Find Bin** with fixed histogram bins then predicts the execution time based on the average runtime measured in the microbenchmark.

If the histogram has variable bins, a binary search algorithm is executed to find the bin. To design a model for this less straightforward case, we investigate the runtime behaviour with different inputs using a microbenchmark. We implement our microbenchmarks using Google’s benchmark framework [13], which simplifies testing for multiple parameter combinations and repeating measurements for stable results. Based on the definition of a binary search, we hypothesise the following:

- H1. The maximum search time is influenced by the size of the search array, as having more bins increases the maximum search distance.
 - The histogram size should be a parameter in our model.
- H2. Some target bins have a shorter search time because of a shorter search distance.
 - The model should estimate the search distance based on the search value.
- H3. The path that is taken to find a bin is deterministic and a bin’s search time, therefore, does not vary significantly over multiple runs.
 - We can model the runtime for n searches of bin b as n times the expected runtime of one search of b .
- H4. The binary search is dominated by memory accesses to the array with bin edges (\neq histogram array).
 - The model should take the cache performance into account.

To test these hypotheses, we implement a microbenchmark that measures the average search time for finding one bin. We experiment with small and large search arrays, ranging from 2^3 to 2^{24} bins containing double-precision values (H1). Additionally, we test with 5 different bin targets, each with a different offset from the start of the array: 0, 0.25, 0.5, 0.75, and 1.0 times the size of the array (H2). To determine the variability, we repeat each search 10000 times (H3).

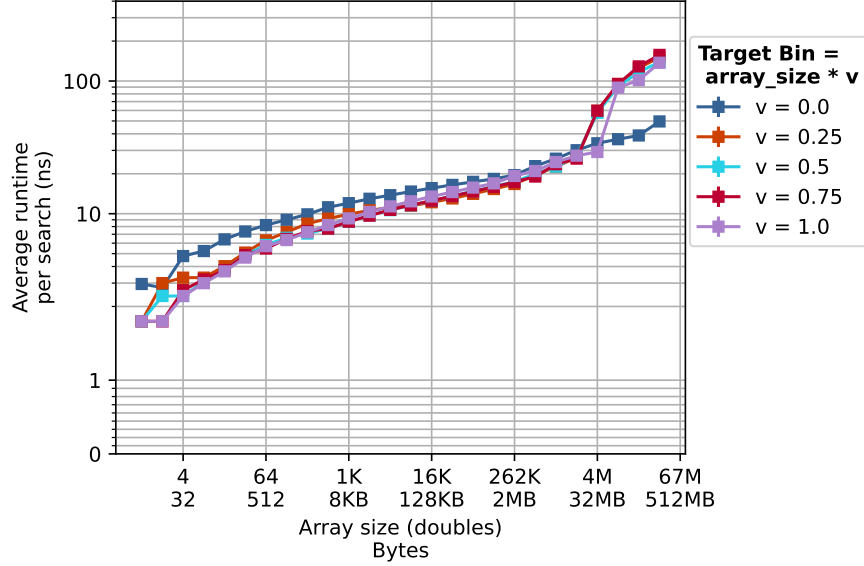


Figure 4.2: Binary search microbenchmarked runtime. The average search time over 10000 iterations of the binary search, repeated 3 times, is plotted with standard deviation error bars.

The result of this microbenchmark is shown in Figure 4.2. As expected, the plot shows that the maximum runtime increases as the search array size increases, which supports H1. Secondly, note that there is little variability in the runtime when searching the same target, which also confirms that the search path is deterministic (H3). However, for arrays containing up to 4M doubles, we do not observe a difference in search time for different target bins (H2). Although, unexpectedly, for array sizes larger than 32MB, the search time for the first element in the array ($v = 0$) was lower than for the other targets. Since the binary search starts in the middle of the array, we would expect the target bin $v = 0.5$ to result in the lowest search time, which was not the case.

To gain an insight on the influence of caching (H4) on the runtimes observed in Figure 4.2, we also use *hardware performance counters*: special-purpose registers that track low-level event such as instructions executed, cache misses, and branch predictions. Using Google Benchmark’s framework, we can collect performance counter data by setting the flag `-benchmark_performance_counters` when running the benchmark. We collect the following events: number of instructions, L1 loads, L1 misses, and L2 misses. From the number of misses, we can infer the number of loads to the immediate lower cache level, i.e., we collect the number of L2 and L3 loads. However, we could not enable performance counters for L3 cache miss events, possibly due to missing privileges or missing hardware support.

4. CPU IMPLEMENTATION

Interlude 4.1: Linear Search

To ensure that the Google Benchmark framework reports the expected performance counters, we replaced the search algorithm in our binary search microbenchmark with a very predictable algorithm – a linear search – for which we can easily anticipate the behaviour in advance: in a linear search algorithm, we iterate over the array with a stride of 1, starting from the beginning, until an element with a value lower than the input value is found. Thus, the runtime, number of instructions, and the number of memory accesses should increase at the same rate as the increase of the offset. Additionally, once the size of the array exceeds the size of a cache, we should start seeing an increased miss rate.

A major challenge we encountered in collecting performance counter data with Google Benchmark, which uses `libpfm` for performance counters, is that documentation on available events is severely lacking. Using the command `perf list`, we can get a list of generic performance counter events and hardware-specific ones. Some listed events, however, such as `l2_cache_accesses_from_dc_misses`, could not be found when running the benchmark. Additionally, most events are missing a description of what is exactly measured. For example, it is unclear which cache level the events `cache-misses` and `cache-references` refer to and whether these include prefetches. Although, some of these unknowns can be reverse-engineered by comparing the collected counts against expected patterns, as shown below.

In the linear search benchmark, we collect the following events: `instructions`, `L1-dcache-load-misses`, `L1-dcache-loads`, `cache-misses`, and `cache-references`. By evaluating at which array sizes the counts for `cache-references` and `cache-misses` increase, we can determine which cache level is measured by this event.

The result of this benchmark is shown in Figure 4.3. The figure plots the collected performance counter data against the array size next to the measured average runtime on a log-log scale. The plots include vertical lines indicating the capacity of the three cache levels of our test system. We can observe that, as expected, the number of executed instructions and the number of L1 cache loads increases linearly with the increase of the array size, except for when $v = 0$. When the destination bin is 0, the performance counters data and

the runtime remains constant at ~ 0 because the search distance is always 1, regardless of the size of the array. Note that by averaging over the number of repetitions, any compulsory misses that occur during initialisation of the data array or the first execution of the search algorithm become negligible and are not shown in the plot; a compulsory miss occurs when a cache line is accessed for the first time and needs to be loaded into the cache.

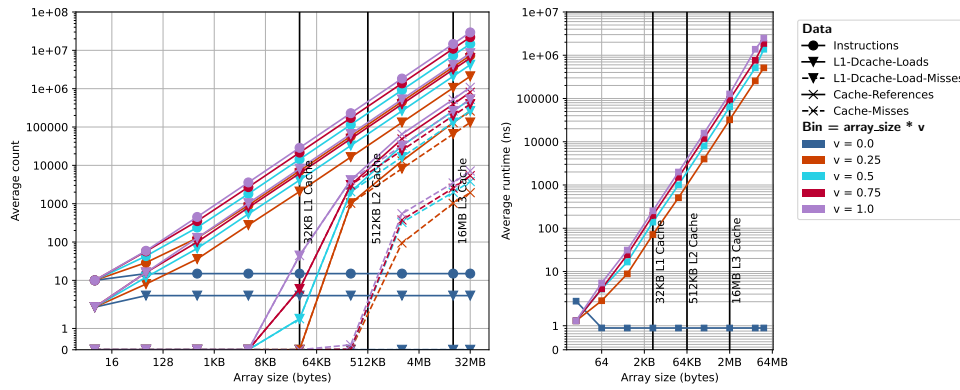


Figure 4.3: Linear search microbenchmark. Average over 1000 searches, repeated 3 times. The left plot contains the collected performance counters and the plot on the right contains the measured runtime.

Additionally, the plot shows that the larger the bin distance, the higher the counts and the longer the runtime for a given array size. When focusing on the performance counter event **cache-misses**, we notice that it is near zero until the capacity of the L2 cache is reached. Also, the count for **cache-references** matches the count for **l1-dcache-load-misses**. This indicates that these events refer to the L2 cache and that it is sufficient to plot the L1 cache misses without losing information about the number of L2 loads. There is a noticeable increase in runtime after exceeding the L2 and L3 cache capacities, which can be linked to the increased L1 and L2 cache misses. In conclusion, the microbenchmark reports the performance counter data that we expect.

4. CPU IMPLEMENTATION

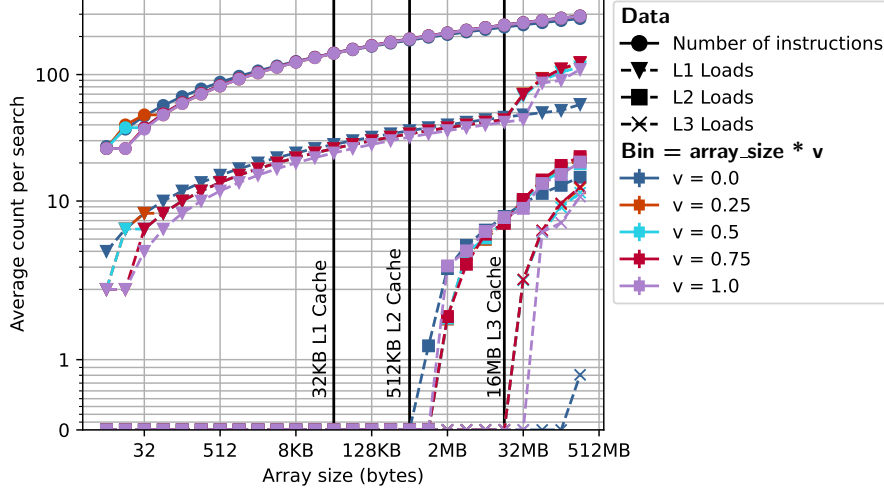


Figure 4.4: Binary search performance counter data. Averaged over 10000 iterations of a binary search, repeated 3 times, with standard deviation error bars.

In Figure 4.4, the collected performance counter data for the binary search benchmark is plotted in a log-log plot. Firstly, we notice that the number of executed instructions and the number of data accesses (L1 loads), are not influenced by the location of the target bin. Although, for array sizes larger than 32MB, there is a jump in the number of L1 loads with search targets other than $v = 0$, even though the number of executed instructions remains the same between different search targets. This is likely an artefact of the benchmark itself. Upon further investigation, ROOT implements a binary search using the C++ standard library method `std::lower_bound`. This function never exits early and always searches the full binary tree, to determine the index in which all values to the left are lower. Note that this method implicitly assumes that the input array is sorted. This ensures that we always fill the ‘first’ suitable bin, even with duplicate bin edges (i.e., zero-width bins). Thus, in contrast to our hypothesis that the search distance differs for different search targets (H2), the binary search always searches $\log_2(n)$ elements, where n is the number of bin edges.

Combining the observed runtimes in Figure 4.2 with the collected performance counter data, we can conclude that the binary search algorithm is indeed dominated by the latency of the memory accesses (H4). Subsequently, we can model the runtime by predicting the number of accesses to the different cache levels. We do this using Khuong and Morin’s cache misses model for a binary search with a sorted array [22]. They note that the binary search benefits from the cache in two ways:

- **Temporal locality:** the binary search always starts at the root node of the (fixed) binary tree and traverses to a leaf. Consequently, the most frequently accessed values are at the top of the tree. To illustrate this effect, Figure 4.5 depicts the bins that are accessed when searching different bins in an array with 64 elements. We can see that the three most accessed bins are 32 (head node), 16 and 48 (nodes at the next level). Consequently, the cache will mostly be filled with frequently accessed values.

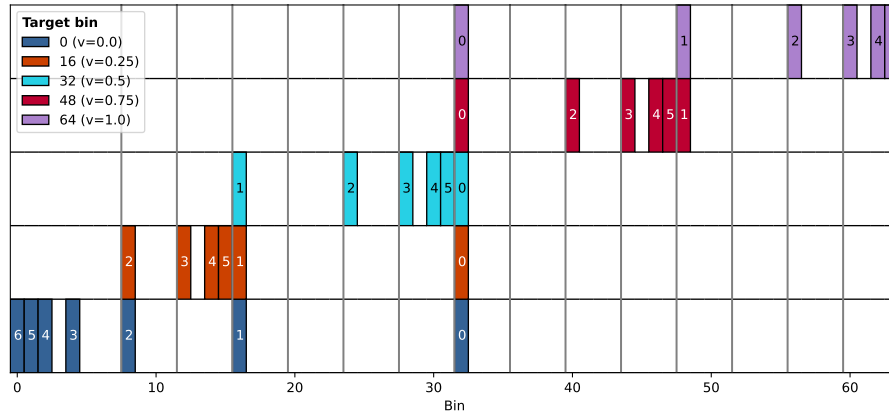


Figure 4.5: Access pattern of `std::lower_bound`. Each row illustrates the pattern for a different target bin in an array with 64 bins. The vertical lines indicate the edges of cache lines that fits 4 data-items.

- **Spatial locality:** once the search range has reduced to a size less than or equal to the number of items that fit in a cache line, the remaining search only results in accesses to one or two cache lines. This can also be observed in Figure 4.5.

Attempt 1: KM model [22]

If the array has n bins, the binary search accesses $\log_2(n)$ elements. Given many searches, we can assume that the cache of size C is filled with the top C elements of the implicit binary tree, so each search accesses $\log_2(C)$ elements that hit in the cache. Therefore, Khuong and Morin suggests that the average number of cache misses can be roughly estimated with the following equation:

$$\log_2(n) - \log_2(C) + 1, \quad (4.1)$$

However, for small arrays, the result is negative, which cannot happen in practice. For our purposes, we clip the result to zero, which gives us the following function to estimate the number of misses in the cache:

$$\max(\log_2(n) - \log_2(C) + 1, 0) \quad (4.2)$$

4. CPU IMPLEMENTATION

In Figure 4.6, we compare the predicted L1 loads ($\log_2(nBins)$), the misses for each cache level (f_{bscm}), and the runtime of the binary search ($T_{binarySearch}$) against our microbenchmarked results. The figure shows that the L1 loads are accurately estimated, but that the number of cache misses are overestimated.

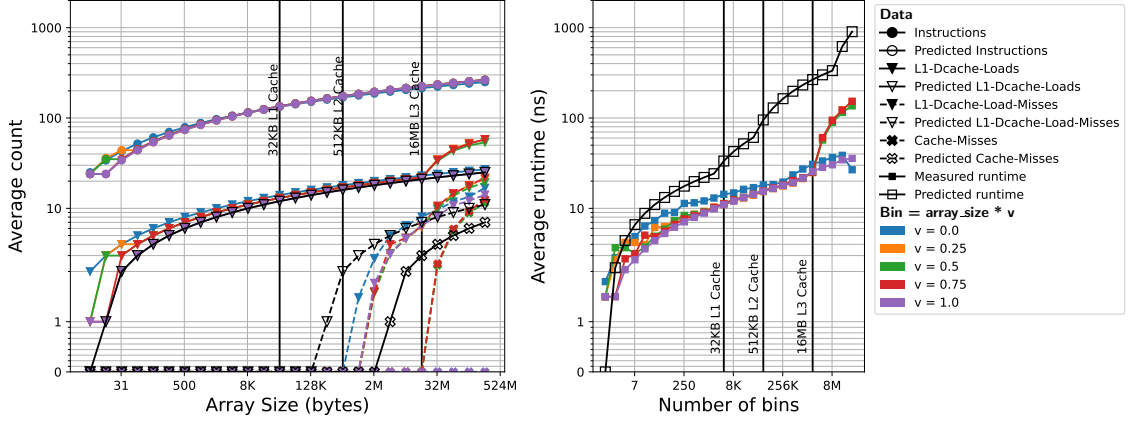


Figure 4.6: Comparison of f_{bscm} and $T_{binarySearch}$ with microbenchmark results.

Microbenchmark results are averaged over 10000 iterations, repeated 3 times.

Equation 4.2 implicitly assumes that all bins of the array are searched with the same frequency, which results in the top C nodes residing in the cache. However, for other distributions where only a subset of the tree is traversed, this is incorrect. For example, in the case of $Normal(0.7, 0.01)$ shown in Figure 3.1b, only one side of the histogram is accessed.

Attempt 2: adapt to data distribution

To account for different data distributions, we modify the equation as follows:

$$\max \left(\log_2(m) - \log_2(C - \log_2 \left(\frac{n}{m} \right)), 0 \right) \quad (4.3)$$

Given a distribution that only results in accesses of m unique elements, where $m < n$, we have $(\log_2(n) + 1) - (\log_2(m) + 1) = \log_2$ elements that are always accessed and likely reside in the cache (see Figure 4.7). Only in the subtree with m leaf nodes is it uncertain which direction is traversed. Here, we take the previous assumption again and assume that the top elements of the subtree reside in the cache. Then, the number of top elements in this subtree that fit in the cache is equal to the cache size C , minus the space that is already occupied by the elements that are always accessed before entering the subtree. Similarly to the initial formula, the number of misses in the subtree can then be estimated as the number of accesses, minus the accesses in the top elements that fit in the cache.

4. CPU IMPLEMENTATION

Taken together, Equation 4.5 estimates the number of sub-bins, but it requires knowledge of the data distribution. While this limitation does not hinder scenario analyses with predefined parameters, datasets with unknown distributions require distribution estimation. One approach is to generate a histogram from a small random subset of the data, fitting various distributions (e.g., δ , normal, uniform) to determine the most suitable one.

$$f_{subBins}(distr, nBins) = \begin{cases} 1 & \text{distr is Constant} \\ \text{bins_in_range}(\mu - 3\sigma, \mu + 3\sigma) & \text{distr is Normal}(\mu, \sigma) \\ nBins & \text{distr is Uniform} \end{cases} \quad (4.5)$$

Using Equation 4.4, 4.5. and the previously made observation that the runtime can be estimated as a linear combination of the memory accesses multiplied by their latency, we can model the runtime of the binary search as in Equation 4.6. Variables that need to be calibrated to the architecture are highlighted in blue.

$$\begin{aligned} T_{BinarySearch}(nBins, distr) &= num_L1_loads * T_{L1} \\ &+ num_L2_loads * T_{L2} + num_L3_loads * T_{L3} \\ &+ num_main_mem_loads * T_{mmem} \\ &= \log_2(s) * T_{L1} + f_{bscm}(nBins, s, L1_size) * T_{L2} \\ &+ f_{bscm}(nBins, s, L2_size) * T_{L3} \\ &+ f_{bscm}(nBins, s, L3_size) * T_{mmem}, \\ s &= f_{subBins}(distr, nBins) \end{aligned} \quad (4.6)$$

In conclusion, this gives us Model 4.2 which makes a distinction between the two cases: fixed bins and variable bins.

Model 4.2: Find Bin

$$\begin{aligned} T_{BinarySearch}(nBins, distr) &= \log_2(s) * T_{L1} + f_{bscm}(nBins, s, L1_size) * T_{L2} \\ &+ f_{bscm}(nBins, s, L2_size) * T_{L3} \\ &+ f_{bscm}(nBins, s, L3_size) * T_{mmem}, \\ s &= f_{subBins}(distr, nBins) \end{aligned}$$

$$T_{FindBinCPU}(binType, nBins, distr) = \begin{cases} T_{Algorithm\ 1:line\ 10} & \text{Fixed bins} \\ T_{BinarySearch}(nBins, distr) & \text{Variable bins} \end{cases}$$

Calibration

In our model, we have 8 parameters that need to be calibrated: the cache capacities ($L1_size$, $L2_size$, and $L3_size$), cache latencies (T_{L1} , T_{L2} , T_{L3} , and T_{mmem}), and the duration of computing the bin with fixed-size bins ($T_{Algorithm\ 1:line\ 10}$).

To calibrate $T_{Algorithm\ 1:line\ 10}$ for fixed bins case, we again utilise a microbenchmark, shown in Listing 4.1¹. The benchmark repeats the computations 1 billion times, for arbitrarily chosen values. To prevent the compiler from optimising away the arithmetic operations, we generate random values for the variables every iteration. By default, the framework measures the runtime of the loop over the state, but by manually setting the iteration time, we exclude the setup of the variables from the timings (see also section B.1).

Listing 4.1: Calibration microbenchmark for variable bins.

```
static void BM_FixedSearch(benchmark::State &state) {
    long long bin;
    long long repetitions = 1e9;

    for (auto _ : state) {
        for (int n = 0; n < repetitions; n++) {
            int nbins = rand();
            double x, xmin, xmax;
            x = xmin = xmax = rand();

            auto start = std::steady_clock::now();
            bin = 1 + int(nbins * (x - xmin) / (xmax - xmin));
            auto end = std::steady_clock::now();
            auto elapsed_seconds = std::chrono::duration_cast<fsecs>(end -
                start);
            state.SetIterationTime(elapsed_seconds.count());
        }

        state.counters["repetitions"] = repetitions;
        state.counters["bin"] = bin;
    }
    BENCHMARK(BM_FixedSearch)->UseManualTime()->Unit(benchmark::kMillisecond);
}
```

For the case with variable bins, we require the latencies for accessing the different memory levels. To obtain these, we ran the `lat_mem_rd` benchmark provided by the LMBench suite using a single thread [25]. This benchmark measures the latency of traversing an array with varying sizes and strides. For arrays that fit in a certain cache level, the accesses will not result in cache misses for that level. The results of the benchmark are shown in

¹Calibration benchmarks can also be found on GitHub: https://github.com/jolly-chen/msc-project/blob/main/microbenchmarks/calibration_microbenchmarks.cu

4. CPU IMPLEMENTATION

Figure 4.8. The capacity of the three cache levels of our AMD processor with sizes 32KB, 512KB, and 16MB, queried using the `getconf -a | grep cache` linux command, are also plotted as vertical lines. From the plateaus shown in the graph, we can infer that we have a latency of 2.2, 7, 25, and 250 nanoseconds for the L1 cache, L2 cache, L3 cache, and main memory respectively.

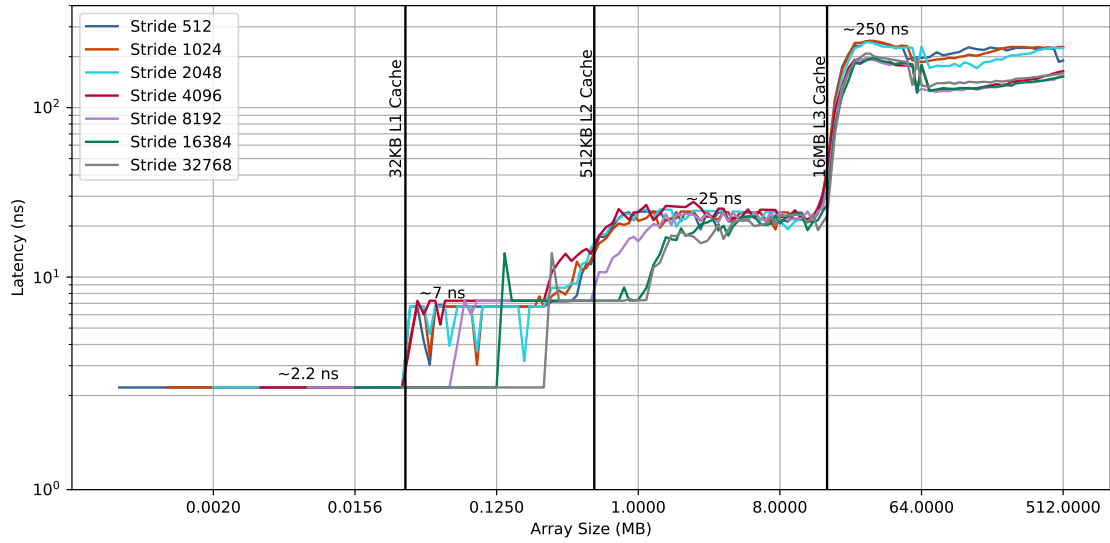


Figure 4.8: LMbench lat_mem_rd results. Memory read latency with a single thread for 1024 different strides and array sizes.

In summary, the parameters can be calibrated step-by-step as follows:

- 1). Run the microbenchmark shown in Listing 4.1 to get $T_{Algorithm\ 1:line\ 10}$.
- 2). Execute the `getconf -a | grep cache` command to query the cache capacities for the parameters ($L1_size$, $L2_size$, and $L3_size$) in bytes.
- 3). Run the `lat_mem_rd` benchmark of LMBench, plot the measured latencies against the array size, and read out the four runtimes plateaus T_{L1} , T_{L2} , and T_{L3} .

Using this calibration process, we have the following values for our test system with an AMD EPYC 7420P CPU:

Calibration 4.1: Find Bin

- $L1_cache_size = 32768$ bytes
- $L2_cache_size = 524288$ bytes
- $L3_cache_size = 16777216$ bytes
- $T_{Algorithm\ 1:line\ 10} = 26.1$ ns
- $T_{L2} = 2.2$ ns
- $T_{L1} = 7$ ns
- $T_{L3} = 25$ ns
- $T_{mmem} = 250$ ns

Validation

Our model distinguishes two cases, fixed and variable bins, which we validate separately for different histogram sizes and distributions. We compare the prediction against benchmarked runtimes of the “Find Bin” component. The prediction is computed as the modelled runtime of “Find Bin” for a single event ($T_{FindBin_{CPU}}$), multiplied by the total number of events (1 billion). The results of this validation are shown in Figure 4.9.

As expected, in the left plot, we can observe that the runtime remains constant for fixed bins, regardless of the data distribution and histogram size. Moreover, the plot shows that the model can accurately estimate the runtime with a deviation of merely $26.5 - 25.8 \approx 0.7$ seconds.

In the right plot with variable bins, we observe that our model predictions also resemble the measured runtimes, except for the point with 10M bins and a uniform distribution. Here, our model heavily overestimates the runtime performance, because it is likely overestimating the number of cache misses in the binary search. In $f_{subBins}$ (Equation 4.5), we assume that for a uniform distribution in the final histogram, every bin is accessed with the same probability. This assumption possibly did not hold for a subset of events.

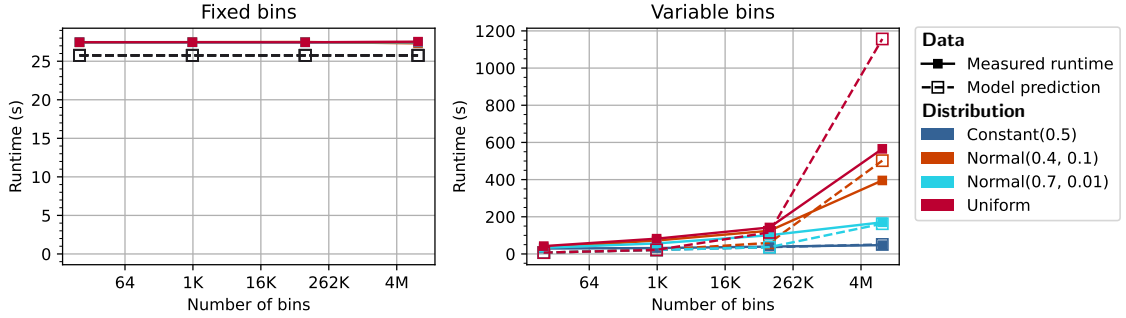


Figure 4.9: Validation of $T_{FindBin_{CPU}}$ prediction with f_{bscm} . Average runtime taken over 5 runs, for 1B events and a bulkszie of 32768.

4. CPU IMPLEMENTATION

4.2 Add Bin Content

The second component in the histogram action is adding the user-specified weight to the relevant bin. The pseudocode for the simplest case, with double-precision input values, is shown in Algorithm 2. The fragment shows that this component involves two operations: reading/writing to the histogram and one addition operation.

Algorithm 2 Pseudocode for “Add Bin Content”

```

1: procedure HISTOGRAM::ADDBINCONTENT(bin, w)
2:   histogram[bin] += w

```

Design

Since memory accesses to the histogram array form the core of this component, we model the time for “Add Bin Content” based on the estimated memory latency, which depends on the cache performance. Similarly to the model for **Find Bin** with variable bins, by estimating the number of misses to each cache level, we can predict the average time of accessing the histogram array as a linear combination of the cache latency and the miss rate:

$$\begin{aligned}
T_{AddBinContent} = & T_{L1} + L1_miss * T_{L2} + L1_miss * L2_miss * T_{L3} \\
& + L1_miss * L2_miss * L3_miss * T_{mmem}
\end{aligned} \tag{4.7}$$

Assuming that we have a large number of events and that the previously filled histogram entries remain in the cache, the bins that are likely located in the cache are the ones that are frequently accessed based on the data distribution. Then, using Equation 4.5 again, we can determine the number of bins m that span this range.

If the range of frequently accessed bins is larger than the capacity C of a certain cache level, there is a chance that we encounter a cache miss. Determining the exact probability for which this occurs is non-trivial, as it requires determining the probability that we access $C - m$ bins within some subrange m . Instead, we use binary values for the miss rates and assume we have a cache miss if the frequently accessed range exceeds the cache capacity. This is described by Equation 4.8:

$$f_{abcm}(m, C) = \begin{cases} 1 & m > C \\ 0 & m \leq C \end{cases}, \tag{4.8}$$

where m is the frequently accessed subrange computed with Equation 4.5 and C the cache capacity.

To summarise, we have the model shown in Model 4.3 for estimating the runtime of the “Add Bin Content” component. The parameters that need to be calibrated to the system are highlighted in blue. Furthermore, the function $f_{subBins}$ is defined previously in Equation 4.5.

Model 4.3: Add Bin Content

$$f_{abcm}(m, C) = \begin{cases} 1 & m > C \\ 0 & m \leq C \end{cases}$$

$$T_{AddBinContent_{CPU}}(nBins, distr) =$$

$$\begin{aligned} & T_{L1} + f_{abcm}(s, L1_size) * T_{L2} \\ & + f_{abcm}(s, L1_size) * f_{abcm}(s, L2_size) * T_{L3} \\ & + f_{abcm}(s, L1_size) * f_{abcm}(s, L2_size) \\ & * f_{abcm}(s, L3_size) * T_{mmem}, \\ & s = f_{subBins}(distr, nBins) \end{aligned}$$

Calibration

In this model, we have 7 parameters that need to be calibrated: the cache capacities ($L1_size$, $L2_size$, and $L3_size$) and memory latencies (T_{L1} , T_{L2} , T_{L3} , and T_{mmem}). The steps to calibrate these parameters are identical to the ones described previously for the “Find Bin” component:

- 1). Execute the `getconf -a | grep cache` command to query the cache capacities for the parameters ($L1_size$, $L2_size$, and $L3_size$) in bytes.
- 2). Run the `lat_mem_rd` benchmark of LMBench, plot the measured latencies against the array size, and read out the four runtimes plateaus T_{L1} , T_{L2} , and T_{L3} .

For the cache sizes and calibration of the cache level latencies, we take the values obtained previously, which gives us the following values:

Calibration 4.2: Add Bin Content

- $L1_cache_size = 32768$ bytes
- $L2_cache_size = 524288$ bytes
- $L3_cache_size = 16777216$ bytes
- $T_{L1} = 2.2$ ns
- $T_{L2} = 7$ ns
- $T_{L3} = 25$ ns
- $T_{mmem} = 250$ ns

4. CPU IMPLEMENTATION

Validation

To validate our model, we measured the runtime of the `Add Bin Content` component for different histogram sizes and distributions. The results of this are shown in Figure 4.10.

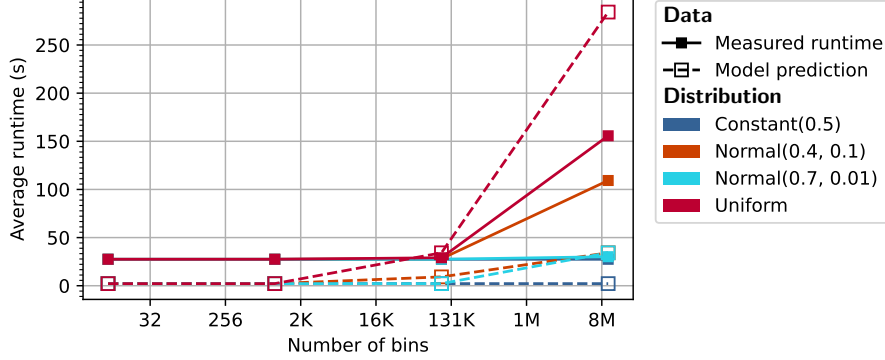


Figure 4.10: Validation of $T_{AddBinContent}$ prediction. Average runtime taken over 5 runs, for 1B events and a bulk size of 32768.

The results show that our model underestimates the runtime for 10 and 1000 bins, but overestimates for 100K and 10M bins. However, according to both the measured runtimes and our model predictions, the runtime performance for the different distributions can be ranked from best to worst as follows: $Constant > Normal(0.7, 0.01) > Normal(0.4, 0.1) > Uniform$. Thus, our model accurately preserves the performance ranking between the different distributions.

Since the L1 cache on our test system has a capacity of 32KB, our model does not expect any cache misses for histograms with up to $\frac{32000}{8} = 4000$ bins with double-precision values. This results in a runtime of $2.2 \text{ ns} * 1\text{B events} = 2.2 \text{ seconds}$. In the plot, the measured runtime is longer than our prediction for both 10 and 1000 bins, and the deviation is the same for both points. This indicates that our model is missing some constant, which could be the addition operation and some overhead incurred by the benchmark timers.

For larger histograms, the predicted runtime increases for the non-constant distributions, because our model expects to have accesses in lower cache levels. When observing our measured runtimes, however, we notice that the runtime remains constant at approximately 25 seconds regardless of the distribution for histograms up to 100K bins. This indicates that our model generally overestimates the cache misses, which could be because we only consider the distribution of the final histogram and not per bulk. If the data is sorted, we could be accessing the same bins more frequently within a bulk which results in a better cache performance, which we do not take into account.

4.3 Update Statistics

The last component is the update of running sum variables to keep track of histogram statistics (mean, standard deviation, and higher momenta). The number of sum variables depends on the dimension of the histogram. The pseudocode for this is displayed in Algorithm 3, where the various sum variables are member variables of the histogram class. Thus, the total number of sum variables that are updated for each event, given an dim -dimensional histogram, is $2 + 2 * dim + \frac{dim * (dim - 1)}{2}$.

Algorithm 3 Pseudocode for “Update Stats”.

```

1: procedure HISTOGRAM::UPDATESTATS( $c, w$ )
2:   if  $c$  not underflow or overflow then
3:      $sum_w \ += w$ 
4:      $sum_{w^2} \ += w * w$ 
5:     for  $i$  in  $0, \dots, dim - 1$  do
6:        $sum_{wc_i} \ += w * c_i$ 
7:        $sum_{wc_i^2} \ += w * c_i * c_i$ 
8:       for  $j$  in  $0, \dots, i$  do
9:          $sum_{wc_i c_j} \ += w * c_i * c_j$ 

```

Design

In our case study, we only analyse one-dimensional histograms, so we update 4 sum variables for each event. Since the number of variables is relatively low, and they are updated for every event, we assume that they remain in the L1 cache. The input coordinates and weights are also assumed to be in the L1 cache or registers, since these variables are also read in the **Find Bin** and **Add Bin Content** components, which are executed before this component. In addition, as the type and quantity of arithmetic operations remain the same, the runtime is likely constant. Similarly to the model for the fixed bins case of **Find Bin**, our model for **Update Statistics** can predict the runtime based on the average runtime measured in a microbenchmark of the component. This results in Model 4.4, where $T_{Alg3-a:b}$ refers to the time it takes to execute the computations from line a to b in Algorithm 3.

Model 4.4: Update Statistics

$$T_{UpdateStats_{CPU}} = T_{Alg3-3:7}$$

4. CPU IMPLEMENTATION

Calibration

In Model 4.4, we have one parameter that we calibrate using a microbenchmark: $T_{Alg3-3:7}$. The implementation of the microbenchmark is shown in Listing 4.2. For completeness, we show the microbenchmark for arbitrary dimensions. However, for our purposes, we only run this benchmark for a single dimension, which gives us a latency of around 5.86 nanoseconds on the DAS-6.

Listing 4.2: Microbenchmark for calibrating $T_{UpdateStats}$

```
static void BM_UpdateStats(benchmark::State &state) {
    long long repetitions = 1e6;
    auto dim = state.range(0);
    std::vector<double> stats(2 + dim * 2 + dim * (dim - 1) / 2, 0);

    for (auto _ : state) {
        double elapsed_seconds = 0;
        for (int i = 0; i < repetitions; i++) {
            auto w = rand();
            std::vector<double> coords;
            for (auto i = 0; i < dim; i++)
                coords.emplace_back(rand());

            auto start = Clock::now();
            int offset = 2;
            stats[0] += w;
            stats[1] += w * w;
            for (auto ci = 0U; ci < coords.size(); ci++) {
                stats[offset++] += w * coords[ci];
                stats[offset++] += w * coords[ci] * coords[ci];
                for (auto cpi = 0U; cpi < ci; cpi++)
                    stats[offset++] += w * coords[ci] * coords[cpi];
            }
            auto end = Clock::now();
            elapsed_seconds += std::chrono::duration_cast<fsecs>(end -
                start).count();
        }
        state.SetIterationTime(elapsed_seconds);
    }
    state.counters["repetitions"] = repetitions;
    state.counters["dim"] = dim;
}
BENCHMARK(BM_UpdateStats)->Args({1})->UseManualTime()
    ->Unit(benchmark::kMillisecond);
```

Calibration 4.3: Update Statistics

- $T_{Alg3-3:4} = 5.86 \text{ ns}$

Validation

In Figure 4.11, we validate the prediction of the total spent in “Update statistics” using $T_{UpdateStats}$, against benchmarked runtimes. The prediction is computed as the modelled runtime of “Update statistics” for a single event ($T_{UpdateStats}$), multiplied by the total number of events (1 billion).

We observe a straight line with measured runtimes, which confirms our assumption that the runtime does not depend on different histogram sizes and data distributions. In addition, we can observe that our prediction line almost perfectly overlaps our measurements, which indicates that our model is highly accurate.

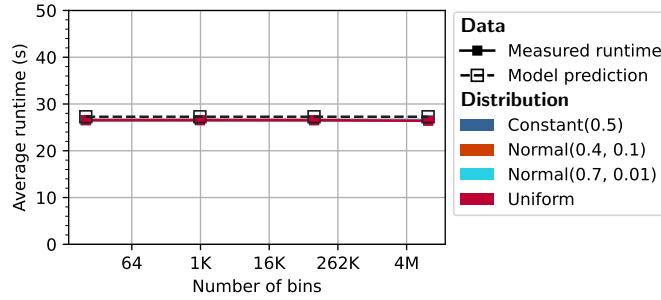


Figure 4.11: Validation of $T_{UpdateStats}$. Averaged over 5 runs with 1B events and bulk size 32768.

4. CPU IMPLEMENTATION

4.4 Total Model Validation

In this subsection, we validate the complete model that combines the models of the individual components. To validate the full model, we run the benchmark shown in Listing 3.1 which contains a single timer around the method that triggers the event loop (see line 20 in Listing 3.1). Thus, unlike in the validation of the components, we do not have any timers around each component.

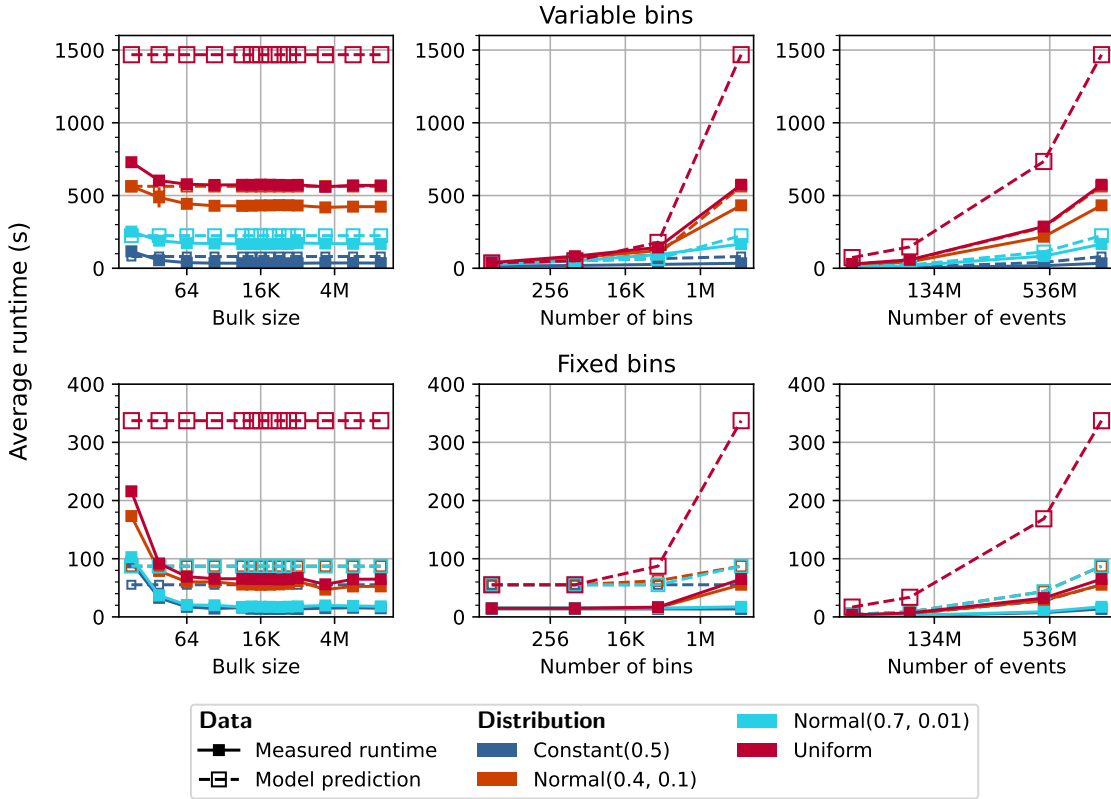


Figure 4.12: Validation of T_{Total_CPU} . Measured runtimes are plotted as the average over 5 runs with standard deviation error bars. The three plots on the top row contain results for histograms with variable bins and the bottom row for fixed bins. In each plot, we vary the parameter labelled on the x-axis and fix the other parameters to the maximum value tested. For example, the plot on the top-left displays the result for different bulk sizes 10M variable-sized bins and 1B events in total. Details on the different data distributions were provided in section 3.4.

In Figure 4.12, we validate our total model with different bulk size, number of bins, and number of events. Focusing on each parameter individually, we observe the following:

- **Bulk size:** the prediction of our model does not differentiate between different bulk sizes. However, from the benchmarked runtimes, we notice that starting from a bulk size of approximately 32 events, the runtime remains constant. Since we parallelise the events within a bulk on the GPU and efficient execution requires much larger workloads, an accurate prediction for small bulk sizes is not necessary for our use case.
- **Type of bins:** the measured runtimes show that the runtime performance is generally worse with variable bins than with fixed bins, which our model accurately captures. However, the model overestimates the execution time by around a factor of 2.5x for variable bins and 4x for fixed bins. The factor being larger for the latter suggests that our model for “Find bin” is less accurate for the fixed bins case, since this is the only component that is affected by the bin type according to our model. However, in our validation of this component (section 4.1, we saw that our prediction for fixed bins was very accurate. This could indicate that the type of bins also influences the other components.
- **Number of bins:** the benchmarked results show that the runtime increases with larger histogram sizes, more so with variable bins than with fixed bins, which our model also captures.
- **Data distribution:** for each parameter, our model accurately ranks the runtime performance based on the distribution. When focusing on the prediction errors, the model’s accuracy based on each distribution can be ranked from best to worst as: *Constant* \rightarrow *Normal*(0.7, 0.01) \rightarrow *Normal*(0.4, 0.1) \rightarrow *Uniform*. This aligns with our estimation for the number of accessed sub-bins from lowest to highest. The higher the number of estimated sub-bins, the more likely we exceed the cache capacity, so the higher our estimated cache misses are. This indicates that our model overestimates the sub-bins and/or punishes the runtime performance for an increase in sub-bins too heavily.
- **Number of events:** the measured runtimes show that the total runtime indeed scales linearly with the number of events (note that the x-axis is logarithmic). This validates our approach of modelling the total runtime by multiplying the predicted runtime for a single event with the total number of events. However, since our model deviates from the measured runtime for a single event, the error between our model and the measured runtimes is magnified with a larger number of events.

4. CPU IMPLEMENTATION

To summarise, our model overestimates the runtime with a factor of up to 4x, likely because we overestimate the quantity or the impact of cache misses. A naive solution to improve the accuracy of our model would be to integrate this factor into our model. However, this factor will be different for other systems, which implies that we have a parameter that needs to be calibrated by comparing the prediction without the parameter against the measured runtime. This would defeat the purpose of modelling, since at this point, one would be better off with simply using the benchmark results for performance comparisons.

For our goal, we accept the deviation, since we aim to determine the cases where we can achieve a speedup of at least 16x (=available CPU cores) with the GPU over CPU execution. On the other hand, the model accurately captures the trends and performance rankings of individual parameters.

Interlude 4.2: Extending the models for multidimensional histograms

While we do not investigate the performance of multidimensional histograms in this work, for completeness, we show how the models can be extended to higher dimensions. The components that are affected by the dimensionality of the histogram are $T_{FindBin}$ and $T_{UpdateStats}$.

$$T_{FindBin}(dim, binType, nBins) = \begin{cases} dim * T_{Algorithm\ 1:line\ 10} & \text{Fixed bins} \\ dim * T_{binarySearch}(nBins) & \text{Variable bins} \end{cases}$$

By inspecting the pseudocode in Algorithm 3, we can determine the number of times the different statistics variables are updated, depending on the histogram's dimension. Based on the observation, we can model $T_{UpdateStats}$ as follows:

$$T_{UpdateStats}(dim) = T_{Alg3-3:4} + dim * T_{Alg3-6:7} + \frac{dim * (dim - 1)}{2} * T_{Alg3-9}$$

To calibrate the three different T variables, we can run the microbenchmark shown in Listing 4.2 for higher dimensions. By fitting the $T_{UpdateStats}(dim)$ function to the measured data points, we can infer the parameter values. Figure 4.13 provides an example of the resulting calibration when using Python's `scipy.optimize.curve_fit` to fit $T_{UpdateStats}(dim)$ to micro-benchmarked data with dimensions between 1 and 128. Note that the `scipy` method can result in negative values for the parameters, which may not be sensible based

on the context. This can be prevented by specifying the lower and upper bounds for the parameters.

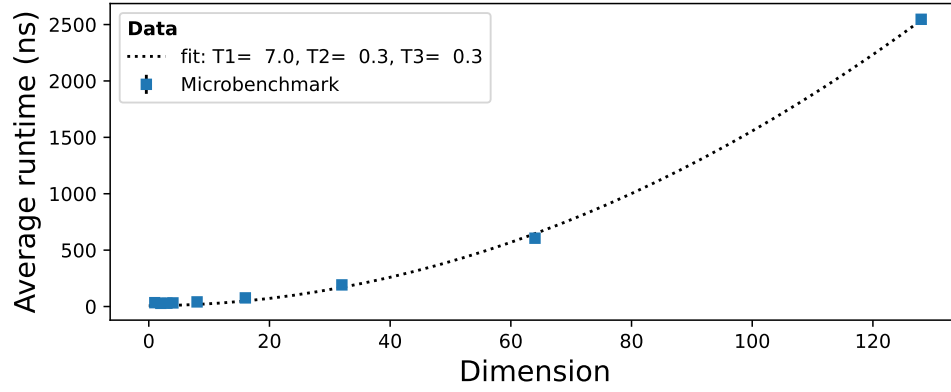


Figure 4.13: Fit $T_{UpdateStats}(dim)$ to microbenchmark data. The benchmark shown in Listing 4.2 was repeated 3 time to get an average.

5

GPU Implementation

In this section, we design the performance model for a GPU implementation of the histogram action. As mentioned in the introduction, RDataFrame does not support GPU execution yet. For the purposes of this study, we implemented basic GPU support for the histogram action. To keep the model simple, the GPU operations are executed in a single CUDA stream, which entails that overlap of computation with communication does not occur. In Chapter 6, we will show how we can use the model to determine implementation directions for hiding memory transfer latencies, to further optimise the runtime performance.

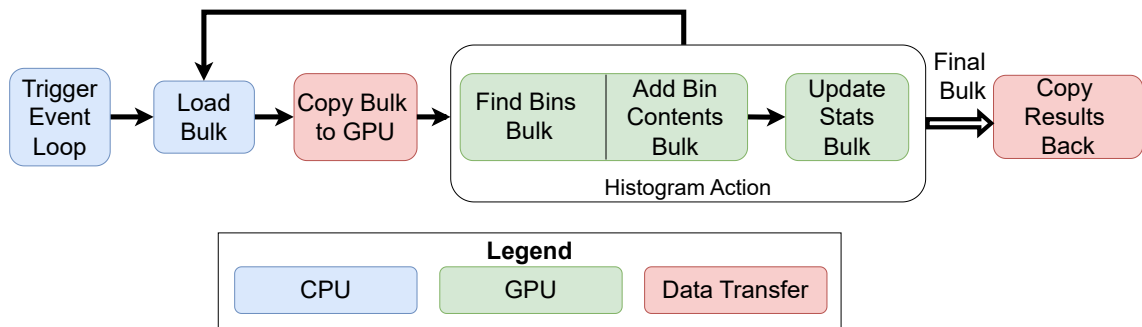


Figure 5.1: Flow diagram of an RDataFrame event loop with one histogram action and GPU support.

Given a single histogram action within an event loop with GPU support, we have the process shown in Figure 5.1. We again loop over the events in bulks, but now we transfer the data to the GPU to process the histogram action on the GPU. Only when all bulks are processed do we copy back the results. Regarding the histogram action, we have the same three components as on the CPU for the histogram action, but the `Find Bin` and `Add Bin Content` components are executed in the same kernel, while `Update Statistics` is processed using a separate kernel.

We parallelise the events within a bulk on the GPU, so we model the performance per bulk instead of per event. In addition, since we do not have asynchronous processing of bulks, we can model the total execution time as a linear combination of the bulk loading overhead, the host-to-device (HtoD) transfer time, and the components' execution times per bulk. We assume that the bulks are always of the same size for the whole duration of the event loop and that the transfer time for copying the histogram results from device-to-host is negligible. To summarise, for the total execution time of the histogram action, we have the model as shown in Model 5.1.

Model 5.1: GPU Synchronous

$$T_{Total_{GPU}} = \underbrace{\left\lceil \frac{nEvents}{bulkSize} \right\rceil}_{\text{Number of bulks}} * \left(T_{LoadBulk_{CPU}} + T_{HtoDBulk} + T_{FindBin_{GPU}} + T_{AddBinContent_{GPU}} + T_{UpdateStats_{GPU}} \right)$$

In the following sections, we describe the performance model for the three histogram action components on the GPU and host-to-device transfers. Again, we exclude the model for $T_{LoadBulk_{CPU}}$ because this constant is also present in the CPU model and therefore does not affect speedup predictions.

5. GPU IMPLEMENTATION

5.1 Find Bin

As described previously in section 4.1, the computations required for determining the bin depend on the type of histogram bins: fixed bins and variable bins. If the histogram has fixed bins, a formula can be used to compute the bin, but for variable-sized bins, a binary search needs to be executed.

In the GPU implementation, the binary search is implemented using `thrust::lower_bound` instead of the `std::lower_bound` method used on the CPU. This replacement was necessary because we do not have access to C++ standard library methods with CUDA. Finding the bins for a bulk of events thus entails having *bulkSize* threads that each run the `thrust::lower_bound` method for a bin coordinate defined by one event.

Design

On the GPU, when we have branching statements, both code paths are executed regardless of the condition result. The incorrect result is then masked afterwards. This is because threads within a warp (group of 32 threads) cannot diverge. This can be avoided by ensuring that there is only divergence between warps and not within warps. However, since the type of bins is determined at runtime since different axes in a multi-dimensional histogram can have different bin types, the CUDA runtime cannot know in advance when a warp does not diverge. As a result, both the formula and the binary search are computed, regardless of the type of bins. For our model, we only model the most expensive case, i.e., variable bins.

Similarly to our process for modelling **Find Bin** with variable bins on the CPU, we microbenchmark the binary search to investigate its performance given different parameters. In this microbenchmark, we launch a kernel in which each thread performs one binary search. The number of searches stays equal to the number of threads, and the number of launched CUDA blocks is set to $\lceil \frac{nThreads}{blockSize} \rceil$. In other words, this is a weak-scaling experiment, where we scale the number of searches with the number of threads. To test the kernel performance for different geometries, we vary the number of threads and the CUDA block size. Additionally, to investigate the algorithmic performance of the GPU binary search, we test with different sizes of the search array and different search destinations. Since the threads are executing the search in parallel, we also alter the search distribution to measure the impact of thread divergence: each thread searches a uniformly random different value (Random), or all threads search for the same array element (Constant). To summarise, we benchmark with the Cartesian product of the following parameter values:

- Block size: 32 (= warp size), 64, 128, 256, 512, 1024 (= maximum supported).
- Array size: powers of 2 in the range [1, 33554432].
- Number of threads: powers of 2 in the range [32, 262144].
- Distribution: Constant(0), Constant(0.5), Constant(1), and Random. With a distribution of Constant(x), each thread searches for the array element that is located at index $x * \text{arraySize}$.

Figure 5.2 displays the benchmarked runtime for different CUDA block sizes. We see that as the bulk size increases, the variance in runtime for different array sizes also increases. Since we do not observe a significant difference in runtime using different block sizes (smaller than 1024), we select a commonly used value – 256 – as the block size for the `FindBin` kernel and only model this instance.

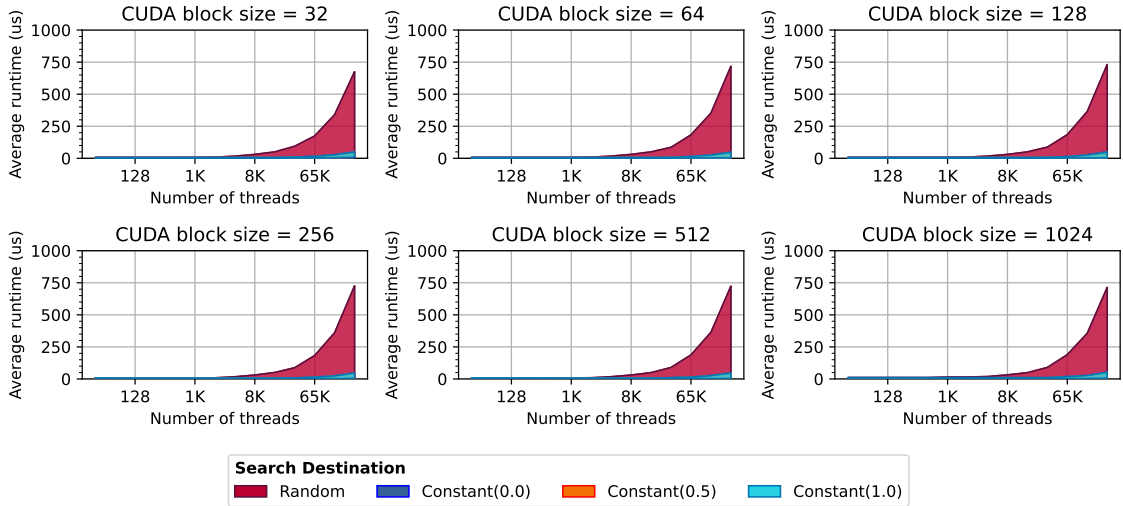


Figure 5.2: GPU binary search microbenchmark results for different CUDA block sizes. For each search destination, the range between the minimum and maximum measured runtime for array sizes 1 to ~33M at each thread count is filled. The runtime is averaged over 1000 repetitions and 3 runs.

If we fix the block size to 256, we have the runtimes shown in Figure 5.3. In the plot, the runtime for different constant search distributions is indistinguishable. This matches the behaviour of the CPU, which again implies that the binary search implementation does not involve any early returns. On the other hand, the performance of the random search noticeably worsens for larger array sizes. This can be attributed to an increase in memory requests since each thread executes a different search path and therefore accesses different array elements. Taken together, this indicates that only the amount of divergence in search paths between threads is relevant, while the actual search target is not.

5. GPU IMPLEMENTATION

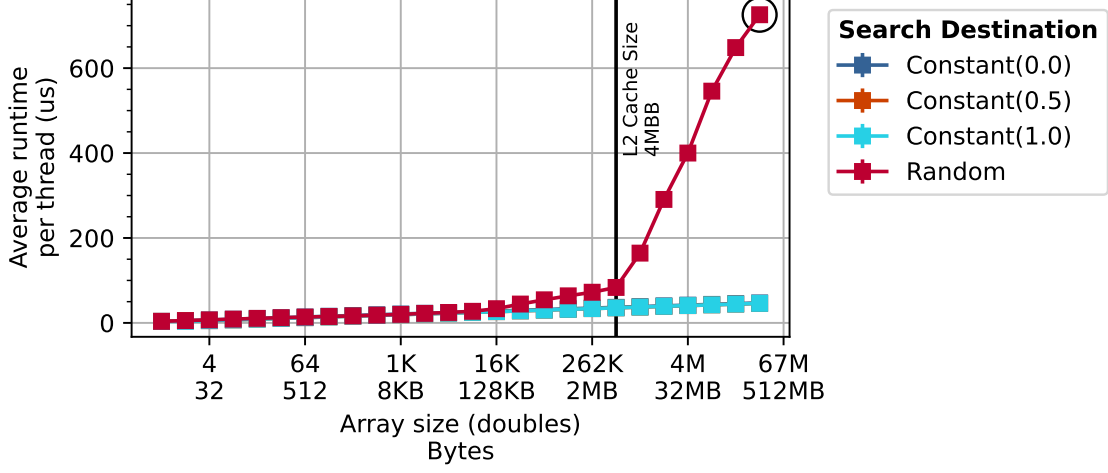


Figure 5.3: GPU binary search microbenchmark results. Runtimes for bulk size of 32768, averaged over 1000 repetitions and 3 runs.

Furthermore, we can observe that – similar to the CPU – the runtime increases for bigger arrays since more elements need to be evaluated to reach the target. Also, we notice that the random distribution resulted in a much longer runtime than the constant distributions. This is likely because a uniform distribution leads to more divergence and thus more memory requests. To determine the amount of divergence based on the distribution, we can use our formula for estimating the number of accessed sub-bins (Equation 4.5).

Interestingly, with the random distribution, we can observe two points where a significant jump in runtime is apparent. This happens at array sizes of approximately 16K and 524K double-precision values. Note that 524K doubles corresponds to ~ 4 MB, which matches the maximum capacity of the L2 cache on the GPU. While we could not query the size of the L1 cache using CUDA’s API, based on the results, we can confidently assume that 16K doubles ≈ 128 KB corresponds to the size of the L1 cache. Thus, a larger search array results in more memory requests and the average latency for the accesses increases if the number of requested addresses exceeds the cache capacity. For our model, this means that we can again predict the runtime based on the cache performance.

Since we have many threads executing in parallel on the GPU, estimating the number of cache misses to each level as we did for the CPU model is non-trivial. Instead, we differentiate between three different regions: sub-bins \leq L1 cache, L1 cache $<$ sub-bins \leq L2 cache, and sub-bins $>$ L2 cache. For each region, we take the maximum observed runtime to get a worst-case prediction.

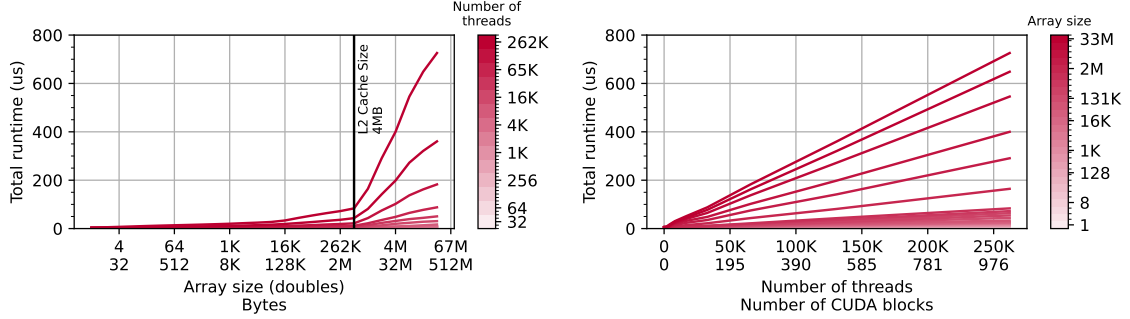


Figure 5.4: GPU binary search microbenchmark results for CUDA block size 256.

Only the runtime results of a random search destination are shown. Each line in the left graph plots the runtime for the thread count specified by its colour shade. In the right graph, each line indicates an array size.

To determine the influence of the thread count on the runtime, we plot the results of the microbenchmark with Random searches for different array sizes and number of threads in Figure 5.4. In the left plot, we notice that when the thread count gets halved, the runtime is also approximately halved. The plot on the right shows that this linear dependency between the runtime and the thread count remains for other arrays. Therefore, based on the thread count t_{fb} used to determine the three different regions, we can scale the prediction using a factor of $\frac{1}{t_{fb}}$.

In conclusion, we have the following as our model for the “Find Bin” component on the GPU:

Model 5.2: Find Bin

$$f_{bsl}(n) = \begin{cases} \lambda_{bsl1} & n < \tau_1 \\ \lambda_{bsl2} & n < \tau_2, \tau_2 > \tau_1 \\ \lambda_{bsl3} & n \geq \tau_2, \tau_2 > \tau_1 \end{cases}$$

$$T_{FindBin_{GPU}}(bulkSize, distr, nBins) = \frac{1}{t_{fb}} * bulkSize * f_{bsl}(f_{subBins}(distr, nBins))$$

Calibration

In Model 5.2, we have 6 parameters that need to be calibrated: the two thresholds τ_1 and τ_2 , the runtime plateaus λ_{bsl1} , λ_{bsl2} , and λ_{bsl3} , and a calibration factor $\frac{1}{t_{fb}}$.

5. GPU IMPLEMENTATION

These parameters can be calibrated step-by-step as follows:

- 1). Microbenchmark the runtime of the binary search kernel with different array sizes and a large number of threads (e.g., 262K). The selected thread count is used as parameter t_{fb} .
- 2). Plot the benchmarked runtimes against the array size, to determine the three plateaus λ_{bsl1} , λ_{bsl2} , and λ_{bsl3} , including the thresholds at which the transitions occur (τ_1 and τ_2). If two jumps cannot be observed, try to increase the thread count.

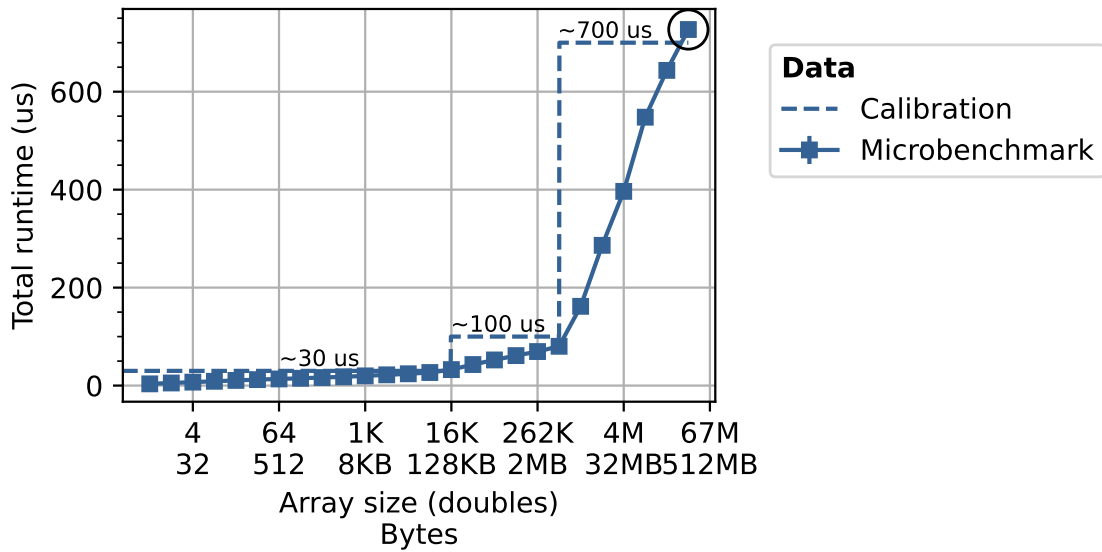


Figure 5.5: GPU binary search microbenchmark results. In the left plot, we have a bulk size of 32768 and on the right, we have 1048576 bins (both data points are circled).

Figure 5.5 shows the results of our microbenchmark and our calibration. The microbenchmark implementation is included in the appendix as Listing C.1. When observing the microbenchmark data, we see two significant jumps at 128KB and 4MB which gives us our two thresholds. The second threshold value should align with the L2 cache capacity, which can be queried using the CUDA API with `cudaDeviceGetAttribute(&out, cudaDevAttrL2CacheSize, deviceNum)` to verify the determined threshold.

To summarise, we calibrated the following values to predict the runtime of the “Find Bin” component in nanoseconds:

Calibration 5.1: Find Bin

- $\lambda_{bsl1} = 30 \mu s$
- $\lambda_{bsl2} = 100 \mu s$
- $\lambda_{bsl3} = 700 \mu s$
- $\tau_1 = 128 \text{ KB}$
- $\tau_2 = 4 \text{ MB}$
- $t_{fb} = 262144$

Validation

Since both the "Find Bin" and "Add Bin Content" components are executed in the same kernel, we could not benchmark these components separately (without a significant overhead). Hence, we validate both components simultaneously when we also have a model for "Add Bin Content" in section 5.2.

5. GPU IMPLEMENTATION

5.2 Add Bin Content

On the GPU, we have two kernel implementations for adding contents to a bin in the histogram: one using solely global memory (Global) and one using shared memory (Local). In both versions, each thread is assigned to one event in the bulk and increments the bin corresponding to its input value.

Algorithm 4 Pseudocode for the GPU implementation of “Add Bin Content” without shared memory (version Global)

```

1: procedure HISTOGRAM::ADDBINCONTENT
2:   for  $i$  in range( $GlobalTid$ ,  $bulkSize$ , step= $nThreads$ ) do
3:      $bin = \text{GetBin} \langle Dim \rangle (...)$  ▷ See Algorithm 1.
4:      $\text{AtomicAdd}(\text{histogram}[i], sMem[i])$ 

```

The pseudocode for the Global version of the kernel is depicted in Algorithm 4. In this component, we model the operation on Line 4; the action of adding the input weight to the calculated histogram bin b_i . Due to potential simultaneous bin increments by multiple threads, an *atomic operation* is required. This ensures that memory access to the bin is performed indivisibly, preventing data corruption or race conditions when multiple threads attempt to access the same memory location concurrently. Depending on the data distribution, this can introduce a significant overhead. In the worst case, all executing threads attempt to concurrently add to the same bin, resulting in high *atomic contention*. To mitigate this, we implemented a more optimised version of the kernel (Local) to reduce the maximum possible contention.

Algorithm 5 Pseudocode for the GPU implementation of “Add Bin Content” with shared memory (version Local)

```

1: procedure HISTOGRAM::ADDBINCONTENT
2:   for  $i$  in range( $localTid$ ,  $nBins$ , step= $blockSize$ ) do ▷ Initialize a local histogram
3:      $sMem[i] = 0$ 
4:   __syncthreads()
5:   for  $i$  in range( $GlobalTid$ ,  $bulkSize$ , step= $nThreads$ ) do ▷ Fill local histogram
6:      $bin = \text{GetBin} \langle Dim \rangle (...)$ 
7:      $\text{AtomicAdd}(sMem[bin], weights[i])$ 
8:   __syncthreads()
9:   for  $i$  in range( $localTid$ ,  $nBins$ , step= $blockSize$ ) do
10:     $\text{AtomicAdd}(\text{histogram}[i], sMem[i])$  ▷ Merge results

```

The pseudocode for version Local of the kernel is shown in Algorithm 5. This kernel performs the filling in two stages:

- 1). Each thread increments a bin in a copy of the histogram stored in shared memory, which is shared only between threads in the same CUDA block. This involves a CUDA block-wide atomic add operation.
- 2). The results in the local histograms are combined into a single histogram stored in global memory. This involves a CUDA device-wide atomic operation.

By splitting the filling into two stages, we decrease the potential contention on the atomic operations. More specifically, the maximum contention in the first stage is equal to the number of threads assigned to a CUDA block (i.e., the block size). In the merging stage, each thread adds the contents of the bin corresponding to their local thread ID to the global histogram, so the maximum contention is equal to the number of blocks (i.e., $\lceil \frac{bulkSize}{blockSize} \rceil$). However, since the available shared memory for storing a copy of the histogram is limited (e.g., 48 KB for NVIDIA RTX A4000), this kernel can only be used when the histogram contains at most $\frac{shared_memory_size_in_bytes}{sizeof(histogram_datatype)}$ bins. By default, we launch the Local kernel, unless the histogram size exceeds the shared memory’s capacity, in which case we utilise the Global kernel instead.

Design

To investigate the performance of the two kernel versions of filling the histogram bins, we perform a similar analysis as for the CPU. We implemented two microbenchmarks, one for each kernel version, that test the runtime performance for different CUDA block sizes, data distributions, histogram sizes, and number of threads. For the data distribution, we tested with two cases that represent the extreme cases for atomic contention:

- 1). *Uniform*, where each thread fills a uniformly random bin, which results in minimal contention.
- 2). *Constant*, where all threads fill the same bin, resulting in maximum contention.

In these microbenchmarks, we perform a weak-scaling experiment: the workload is scaled with the increase in resources. Meaning, that the total number of values that are filled into the histogram remains equal to the thread count, as each thread fills exactly one value.

The results for different CUDA block sizes are displayed in Figure 5.6. We observe that using a block size of 256 results in the most stable results (the least coloured areas), for different bulk sizes and data distributions. Thus, we only model the runtime of the kernel instances with 256 as the block size.

5. GPU IMPLEMENTATION

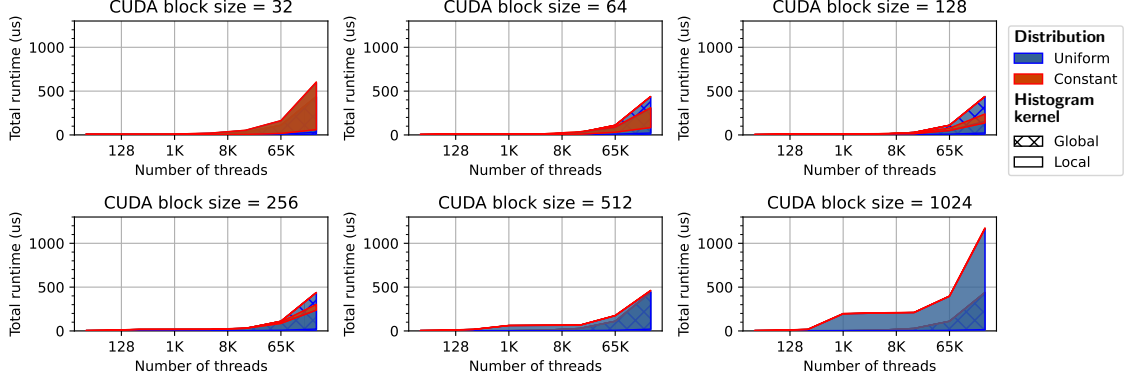


Figure 5.6: GPU Add Bin Content microbenchmark results for different CUDA block sizes. For each kernel version and distribution combination, the range between the minimum and maximum measured runtime for histogram sizes 1 to ~ 33 M at each thread count is filled. The runtime is averaged over 1000 repetitions and 3 runs.

In Figure 5.7, we focus on the runtime results for a CUDA block size of 256 and different histogram sizes, thread counts, kernel versions, and data distribution. Note that we only have runtimes for the Local kernel version up to a size of 32KB, because that is the largest histogram that can fit in the shared memory of our test system. For larger sizes, we can only run the Global kernel.

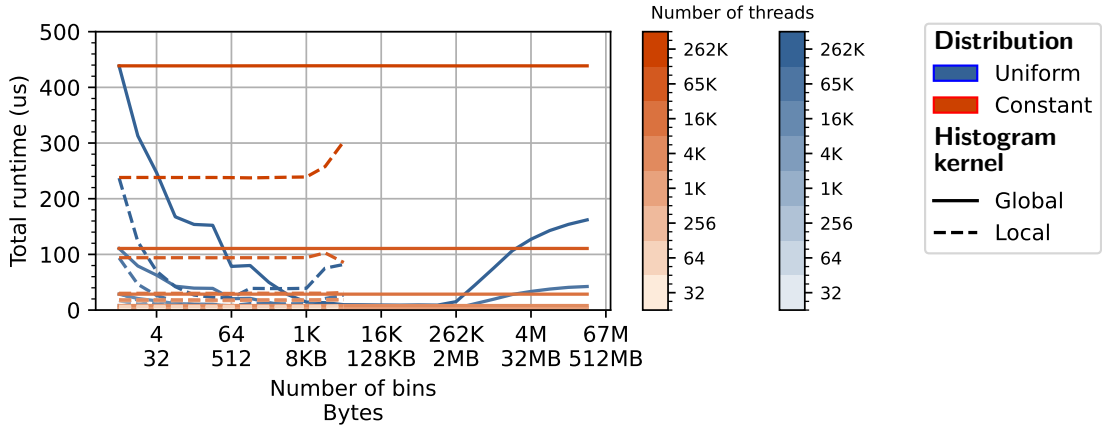


Figure 5.7: GPU Add Bin Content microbenchmark results with CUDA block size 256 for different histogram sizes. Each line in the graph plots the runtime for the thread count specified by the colour bar legend.

Focusing on the uniform distribution (blue lines), we observe that with the Global kernel (dotted lines), the runtime decreases as the histogram size increases until around 4MB. After this point, the runtime increases again and then stabilises. The initial decrease is

likely due to decreased contention on the atomic operations because the probability that multiple threads fill the same bin lowers with a larger number of bins. For sizes larger than 4MB, the histogram does not fit in the L2 cache, so the performance worsens due to an increase in cache misses. For the Local kernel (solid line), we see a similar decreasing trend, but the starting point is at a lower runtime.

For the constant distribution (orange lines), the runtime remains constant across a different number of bins and kernel versions. With this distribution, we are always filling the same bin, so we do not see an effect of cache misses as the number of accesses remains unchanged. The benchmark is essentially measuring the average runtime with maximum contention, which explains why, at 1 bin, the runtime is identical to the runtime with the uniform distribution for the same thread count. In addition, these results confirm that the Local version performs better due to less atomic contention.

Lastly, for the Local kernel, we notice an increase in runtime for both distributions with sizes larger than approximately 1K bins. This is because the number of blocks that can run concurrently on a single SM decreases with increased shared memory usage. In fact, the maximum shared memory per SM is equal to the threshold per block, so if a kernel is launched with the maximum shared memory setting, it only executes a single block per SM. This decrease in parallelism results in an increase in runtime. To sum up, we have observed the following influences on the runtime in the order of most to least impact: atomic contention, cache misses, and level of parallelism. For our initial model, we only consider the first aspect – atomic contention.

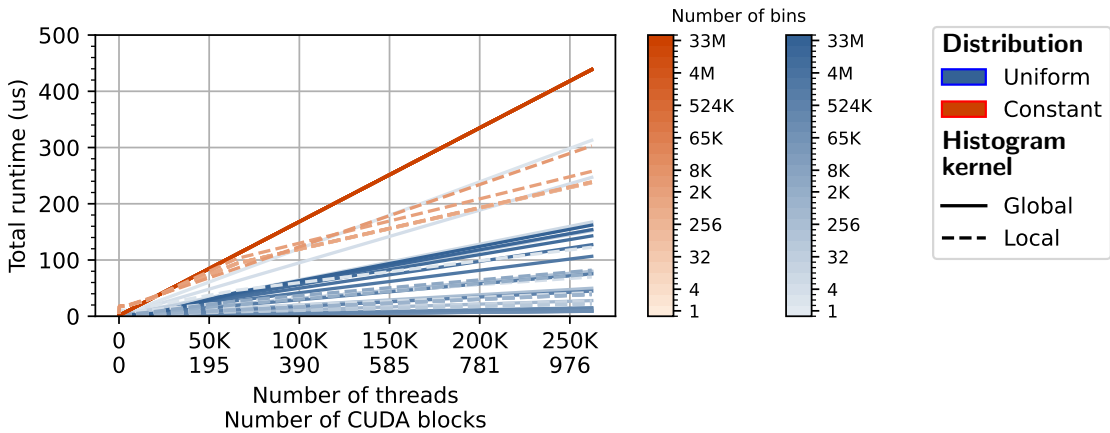


Figure 5.8: GPU Add Bin Content microbenchmark results with CUDA block size 256 for different thread counts. Each line in the graph plots the runtime for the number of bins specified by the colour bar legend.

5. GPU IMPLEMENTATION

If we swap the parameter on the x-axis (number of bins) with the parameter on the colour bar (number of threads), we get the plot shown in Figure 5.8 with linear axes. Here, we can clearly observe a linear correlation between the number of threads and the total runtime. Thus, if we take the data with a constant distribution, we can fit a linear equation of the form $f(x) = \alpha x + \beta$ through the points which can predict the runtime of this component based on the average contention at x threads (i.e., bulk size).

To conclude, our model for the runtime performance of the “Add Bin Content” component on the GPU is shown in Model 5.3. The calibration parameters are highlighted in blue. Furthermore, the function $f_{subBins}$ is defined previously in Equation 4.5.

Model 5.3: Add Bin Content

$$T_{atomic}(nBins, x) = \begin{cases} \alpha_{ag}x + \beta_{ag} & nBins > \text{sharedMemory} \text{ (Global)} \\ \alpha_{al}x + \beta_{al} & nBins \leq \text{sharedMemory} \text{ (Local)} \end{cases}$$

$$T_{AddBinContent_{GPU}}(bulkSize, distr, nBins) = T_{atomic}\left(nBins, \frac{bulkSize}{f_{subBins}(distr, nBins)}\right)$$

Calibration

In our model we have 5 parameters that need to be calibrated: the slopes (α_{ag} and α_{al}), the initial values (β_{al} and β_{ag}), and the capacity of the shared memory per thread block. The latter can be queried using the CUDA API with the method `cudaDeviceGetAttribute(&out, cudaDevAttrMaxSharedMemoryPerBlock, deviceNum)`. The parameters for the linear equations can be calibrated step-by-step as follows:

- 1). Microbenchmark the Global and Local histogram kernels with a range of total number of threads, where every thread fills the same bin.
- 2). Fit a linear equation of the form $f(x) = \alpha x + \beta$, where x is the number of threads, $f(x)$ the measured runtime, and α and β the parameters we want to find. This function approximates the runtime given a contention of x threads on average.

Figure 5.9 shows the result of this calibration process for our test system with an NVIDIA A4000 GPU. The implementation of the microbenchmark that we used is included in the appendix (Listing C.2). The plot shows that our fitted linear equation can accurately describe our microbenchmarked runtimes for the Global and Local versions of the histogram kernel.

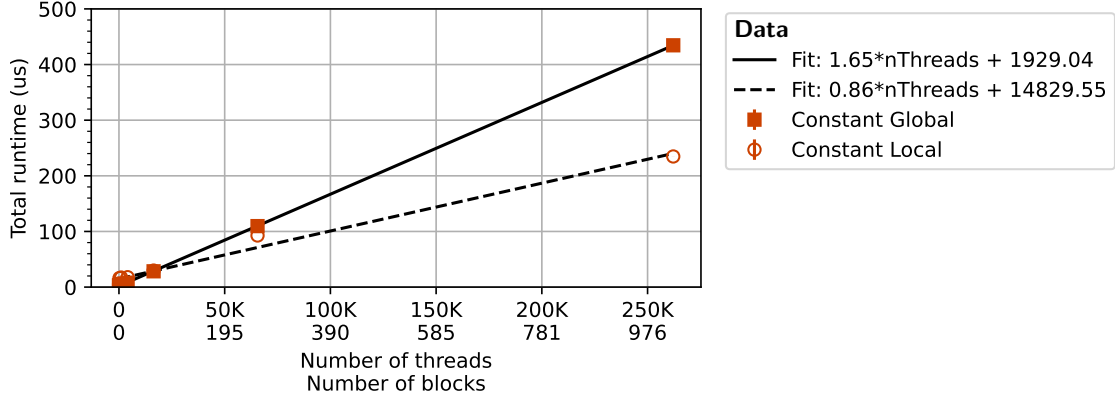


Figure 5.9: Calibration of atomic contention. The data points are taken from Figure 5.8 with 33M bins. The parameters that need to be calibrated to the system are highlighted in blue. Furthermore, the function $f_{subBins}$ is defined previously in Equation 4.5. The linear equation was fitted using Python’s `scipy.curve_fit` method.

To summarise, we use the following values for our calibration parameters to predict the runtime of the “Add Bin Content” component in nanoseconds:

Calibration 5.2: Add Bin Content

- $\alpha_{ag} = 1.65$
- $\alpha_{al} = 0.86$
- $\beta_{ag} = 1929.04$
- $\beta_{al} = 14829.55$
- $sharedMemory = 48 \text{ KB}$

Validation

As mentioned previously, the GPU executes both the “Find Bin” component and the “Add Bin Content” component in the same kernel. Now that we have a performance model for both components, we can validate the models against the measured kernel runtime. Figure 5.10 compares the model predictions against the measured runtimes using NSight Systems for different histogram sizes, bulk sizes, and data distribution.

Firstly, focusing on the histogram size parameter, we notice that the influence of the histogram size varies for different distributions. With the constant distribution, the runtime increases when the number of bins grows up to 100K bins, but for 10M bins there is a slight decrease in runtime. When the spread of the distribution expands, the trend changes to shorter runtimes with small histograms and longer runtimes for large histograms. Note that our model accurately captures this interaction.

5. GPU IMPLEMENTATION

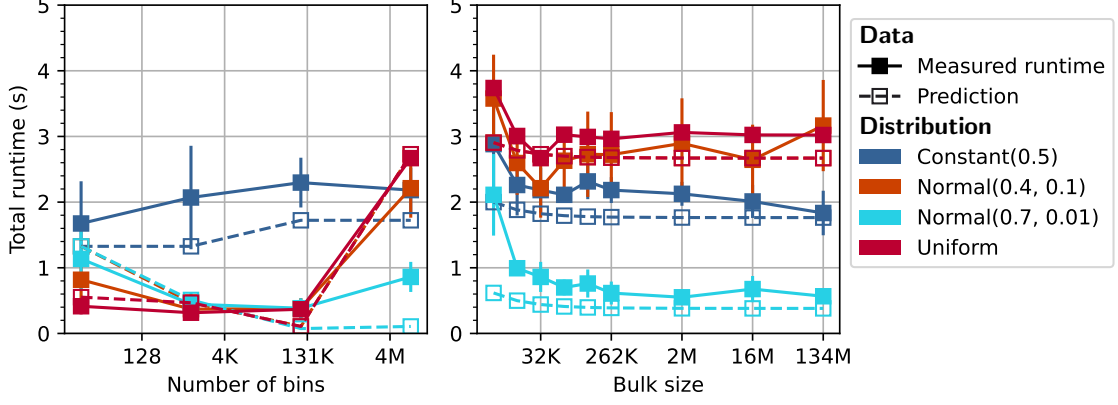


Figure 5.10: Validation of $T_{FindBin_{GPU}}$ and $T_{AddBinContent_{GPU}}$. Measured runtimes are plotted as averages over 5 runs with standard deviation error bars. In the left plot, we have a bulk size of 32K and on the right, we have 10M bins. Runtimes are measured for 1B events in total.

When evaluating the measured runtimes for different bulk sizes, we observe that the runtime decreases for larger bulk sizes. This is expected since more events are processed in parallel when the bulks are larger, and we have less kernel launch overhead because the 1B events are split into fewer bulks.

5.3 Update Statistics

In this section, we describe the GPU model for updating histogram statistics, which involves maintaining running sum variables. This process resembles *transform-reduce* operations, where each element undergoes a *transformation* (such as squaring or multiplication) and the results are aggregated/*reduced* into a single value. In our case, the operation involves squaring the bin weights or multiplying them with coordinate values, followed by a summation.

Since transform-reduce is a common pattern, the CUDA toolkit also offers an optimised kernel in the Thrust C++ template library. We considered using `thrust::reduce`, however, a significant drawback of the parallel Thrust methods is their default fork-join behaviour. This implies that after every reduction operation, the device synchronises with the host because the result is copied back. This would block future optimisations such as overlapping of computations in different bulks.

Algorithm 6 Pseudocode for the transform-reduction kernel

```

1: procedure TRANSFORMREDUCE(in, out)
2:    $r = 0$ 
3:   while  $i < \text{bulkSize}$  do
4:      $r += \text{in}[i]$ 
5:     if  $i + \text{blockSize} < \text{bulkSize}$  then  $r += \text{in}[i + \text{blockSize}]$ 
6:    $r = \text{warpReduceSum}(\text{mask}, r)$  ▷ Reduce within warps
7:   if  $\text{localTid} \% 32 == 0$  then  $\text{sMem}[\text{localTid}/32] = r$ 
8:   __syncthreads()
9:    $\text{shmemExtent} = (\text{blockSize} / 32) > 0 ? (\text{blockSize} / 32) : 1;$ 
10:   $\text{ballotResult} = \_\_\text{ballot\_sync}(\text{mask}, \text{localTid} < \text{shmemExtent});$ 
11:  if  $\text{tid} < \text{shmemExtent}$  then ▷ Reduce final warp
12:     $r = \text{warpReduceSum}(\text{ballotResult}, \text{sdata}[\text{tid}]);$ 
13:  if  $\text{localTid} == 0$  then  $\text{out}[\text{blockIdx}] += r$  ▷ Write result to global memory

```

Consequently, to compute the weighted sums, we implemented a reduction kernel that computes the sum of an array in a tree-based manner. The implementation is based on a sample provided by the CUDA toolkit¹, but has been modified to accommodate for a transformation operation. The pseudocode for this transform-reduce kernel is shown in Algorithm 6.

¹https://github.com/NVIDIA/cuda-samples/blob/e8568c417356f7e66bb9b7130d6be7e55324a519/Samples/2_Concepts_and_Techniques/reduction/reduction_kernel.cu#L431

5. GPU IMPLEMENTATION

Design

Similarly to the execution on the CPU, updating the statistics for a single event has a constant runtime and does not depend on the data distribution or the histogram characteristics. However, on the GPU, we compute the statistics for a bulk of events in parallel. Based on the kernel block size and the number of threads we launch ($\frac{bulksize}{2}$), the execution can be scheduled differently. To determine the influence of these two parameters on the runtime, we microbenchmarked the transform-reduction kernel.

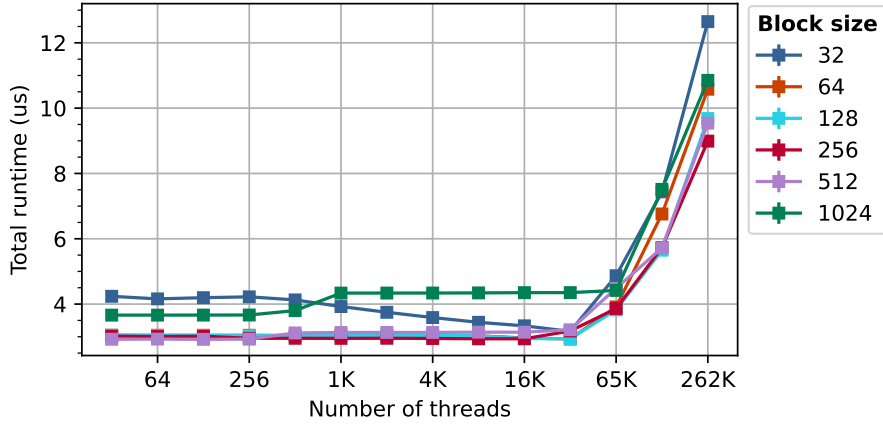


Figure 5.11: Transform-reduce kernel microbenchmark results. The runtime is plotted as an average over 10000 runs, repeated 3 times, with standard deviation error bars. The number of values that are reduced is equal to the twice number of threads.

Figure 5.11 displays the microbenchmarked with a range of bulk sizes and supported block sizes. Here, we again notice that the optimal block size is 256, so we only model the kernel instance with 256 threads per block. Also, the plot displays an exponential trend for a logarithmic increasing thread count, which indicates that the runtime of the transform-reduce kernel scales linearly with the bulk size. Therefore, we can model the runtime of a single reduction kernel by fitting a linear equation to the microbenchmark data. To get a prediction for the total time spent on updating histogram statistics on the GPU, we multiply the prediction for a single transform-reduction with the number of statistics we keep track of (4 in the case of a one-dimensional histogram).

In conclusion, our model for the “Add Bin Content” component on the GPU is shown in Model 5.4. The parameters that need to be calibrated to the system are highlighted in blue.

Model 5.4: Update Statistics

$$T_{UpdateStats_{GPU}}(bulkSize) = \alpha_{usb} * bulkSize + \beta_{usb}$$

Calibration

In this model, we have two parameters that need to be calibrated: α_{usb} and β_{usb} . These can be calibrated step-by-step as follows:

- 1). Microbenchmark the transform-reduce kernel with an increasing number of values as input.
- 2). Fit a linear equation of the form $f(x) = ax + b$ to the results, where x is the number of values to reduce, and $f(x)$ the measured runtime.

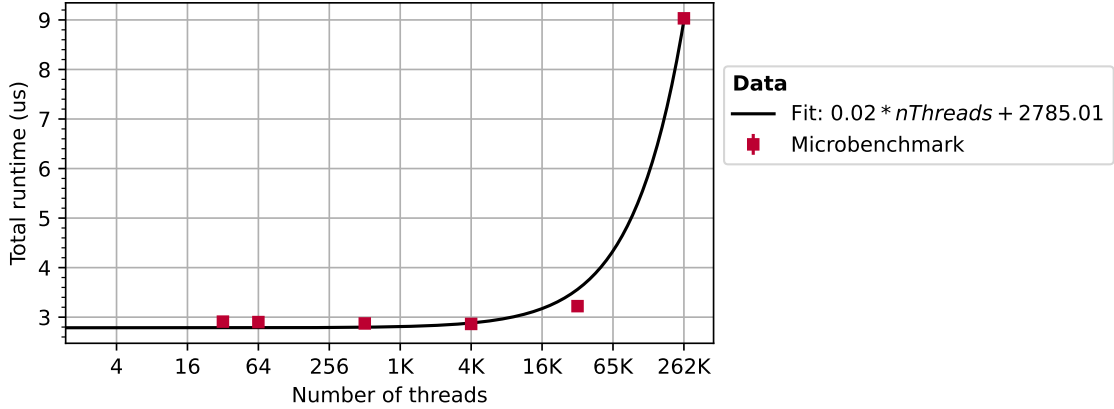


Figure 5.12: Calibration of transform-reduce latency. The number of transform-reduced elements scales with the number of threads.

Figure 5.12 shows the result of this calibration process on our test system with an NVIDIA A4000 GPU. The used microbenchmark implementation is included in the appendix (Listing C.3). We can see that the linear fit closely matches our microbenchmarked data. To summarise, we use the following values for our calibration parameters to predict the runtime of the “Update Statistics” component in nanoseconds:

Calibration 5.3: Update Statistics

- $\alpha_{usb} = 0.02$
- $\beta_{usb} = 2785.01$

5. GPU IMPLEMENTATION

Validation

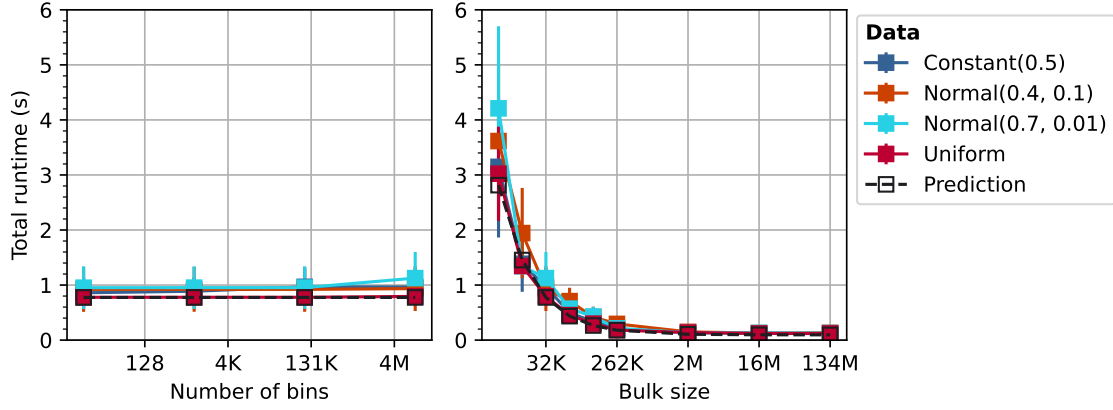


Figure 5.13: Validation of $T_{UpdateStats_{GPU}}$. Average runtime taken over 5 runs for 1B events.

In Figure 5.13, we validate our model for “Update Statistics” component on the GPU. The results show that the performance of this component is indeed not influenced by the number of histogram bins and the data distribution. In addition, our model can highly accurately predict the runtime for different bulk sizes.

5.4 Memory Transfers

As shown in Figure 5.1, we need to transfer bulks of event data from the CPU to the GPU to process them on the device. Since this operation occurs within the event loop for every bulk, it likely results in a non-negligible overhead. In this section, we model the runtime of host-to-device transfers for different bulk sizes.

Design

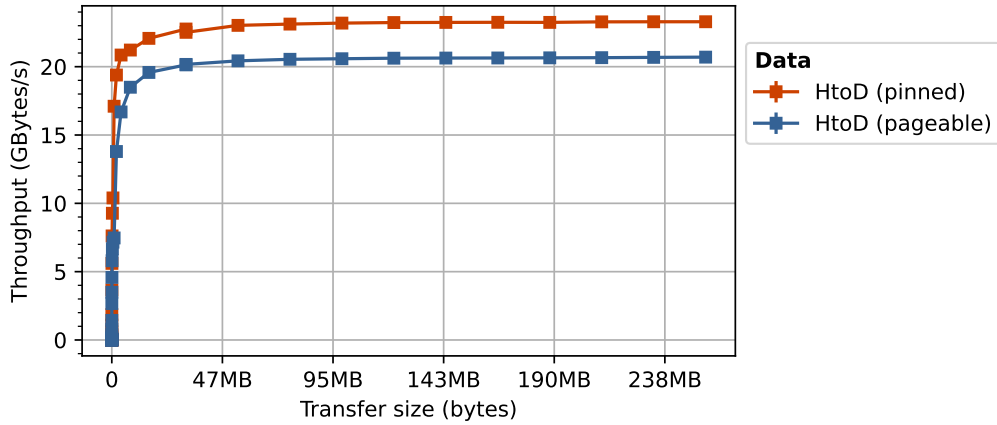


Figure 5.14: Microbenchmarked host-to-device (HToD) transfer throughput for one transfer. Averaged over 300 transfers, repeated 3 times, with standard deviation error bars.

We implemented a microbenchmark that measures the host-to-device transfers for two types of memory allocations: *paged* and *pinned*. By default, CUDA memory allocations are paged, which entails that the allocations can be swapped in and out of physical RAM from/to disk by the operating system. All allocations in the current implementation use paged allocations. For asynchronous execution of data transfers, the CUDA runtime requires pinned allocations, where allocations are page-locked so they are not relocated. Therefore, to make predictions with asynchronous transfers for overlapping communication with computation, we also model the runtime for pinned data transfers.

Our benchmark measures the runtime over 1000 transfers and repeats this 3 times, to get the average transfer time. In Figure 5.14, we display the throughput in gigabytes per second at different transfer sizes. The plot shows that the throughput increases for larger transfer loads, until approximately 48MB, where the throughput stabilises at a rate of around 24 GB/s. Note that this is only 10% of the theoretical memory bandwidth (448.1 GB/s), because the host waits until a transfer has completed before starting the next transfer (i.e.,

5. GPU IMPLEMENTATION

synchronous execution). In practice, multiple transfers can be executed in parallel because the PCIe connection bus that handles the transfers consists of multiple lanes, which can further increase the throughput. This does not occur in the implementation that we model, since it only contains synchronous transfers. In conclusion, our model should predict the total memory transfer time based on the size of the data and the number of transfers.

Attempt 1: lambda-model [31]

To model the memory transfer throughput, we initially attempted to use the “ λ -Model” proposed by Riahi et al. [31]. This is an empirical model that predicts the transfer time using the following equation:

$$T(d) = \alpha + \lambda d, \quad (5.1)$$

where d is the size of the data to transfer in bytes, α the fixed overhead representing the latency of sending the first byte, and λ a transmission coefficient which can be interpreted as the inverse throughput in seconds per byte at data size d . This coefficient is defined as $\lambda = \frac{t-\alpha}{s}$, where t is the measured transfer time at a selected transfer size of s bytes. However, since the throughput differs based on the transfer size, it is not sufficient to select a single size s to determine the coefficient. Instead, the λ -coefficient is computed separately for data sizes smaller and larger than the threshold at which the throughput stabilises.

For larger values than the threshold, which is determined by visually inspecting the bandwidth chart, they use a micro-benchmark. This measures, for k number of selected data sizes, the transfer time. The λ -value is then computed by taking the average:

$$\lambda = \frac{1}{k} \times \sum_{i=1}^k \frac{t_i - \alpha}{s_i}, \quad s_i > threshold, \quad (5.2)$$

where $t_{n_{deg}}$ is the n_{deg} -th measured transfer time for the n_{deg} -th selected data size $s_{n_{deg}}$.

For smaller values, the process is more involved because they use a regression model where they fit the data to a function. As regression input, they use λ values for a selected number of measured points at data sizes smaller than the threshold. The output is a polynomial function of the form

$$\lambda(d) = a_{n_{deg}} d^{n_{deg}} + a_{n_{deg}-1} d^{n_{deg}-1} + \dots + a_1 d^1 + a_0, \quad (5.3)$$

where n_{deg} is the degree of the polynomial, $a_{n_{deg}}$ the n_{deg} -th parameter, and d the transfer size in bytes. However, with our microbenchmarked data, we get the result shown in Figure 5.15.

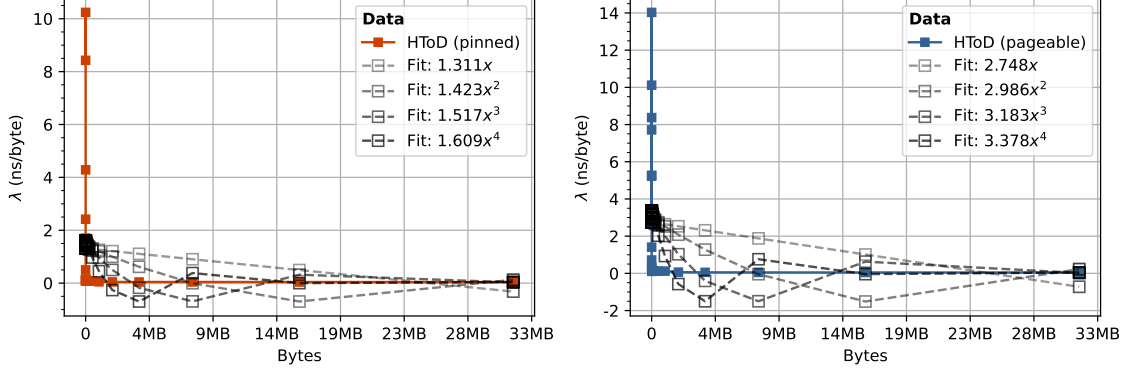


Figure 5.15: Calibration of lambda for data sizes smaller than the threshold. $\alpha \approx 5903$ ns is used for pinned transfers and $\alpha \approx 3951$ ns for pageable transfers in the computation of λ . The function descriptions of the different fits exclude parameters that are zero as a floating point number.

The figure shows the λ values derived from our microbenchmarked data together with 4 different polynomial orders. For both the pinned and pageable transfers, we see that a polynomial functions result in a poor fit of the data because they cannot capture sharp decreases.

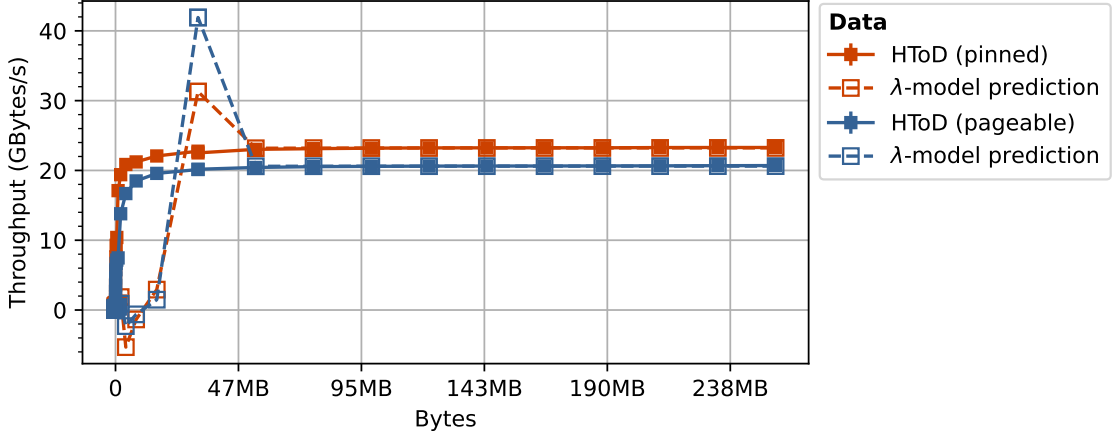


Figure 5.16: Comparison of λ -model predictions with microbenchmarked data. The λ -predictions use the second-order fitted polynomials in Figure 5.15.

When using the calculated λ values to compute Equation 5.1, we get the predicted throughput as shown in Figure 5.16, overlaid with our measured throughput. This further confirms that a polynomial is unsuitable, as it both undershoots and overshoots. In addition, the fit can result in negative values, which does not occur in practice. However, using an average for the inverse bandwidth does result in an accurate fit for larger data transfers.

5. GPU IMPLEMENTATION

The paper that proposed the λ -model likely showed better results, because they tested on a NVIDIA 8600M released in 2011, which is majorly outdated and is far less powerful than our A4000 GPU.

Attempt 2: Gaussian error function

Since a polynomial is shown to be unsuitable for modelling small transfers, we experimented with another function for the regression model. Moreover, we simplify the calibration process by using the measured throughput as input for the regression instead of λ , since the value of α can be captured within the regression method. The alternative option we investigated was using a Gaussian error function $Erf(x)$, which is defined by the integral $Erf(x) = \frac{2}{\pi} \int_0^x e^{-t^2} dt$ and gives an S-shaped curve. Using a regression model, we fit a function of the form:

$$f_{tp}(d) = \alpha_{fp} * Erf((d - \beta_{fp}) * \gamma_{fp}) + \delta_{fp}, \quad (5.4)$$

where α_{tp} , β_{tp} , γ_{tp} , and δ_{tp} are the parameters we want to find. This gives us the improved fit shown in Figure 5.17.

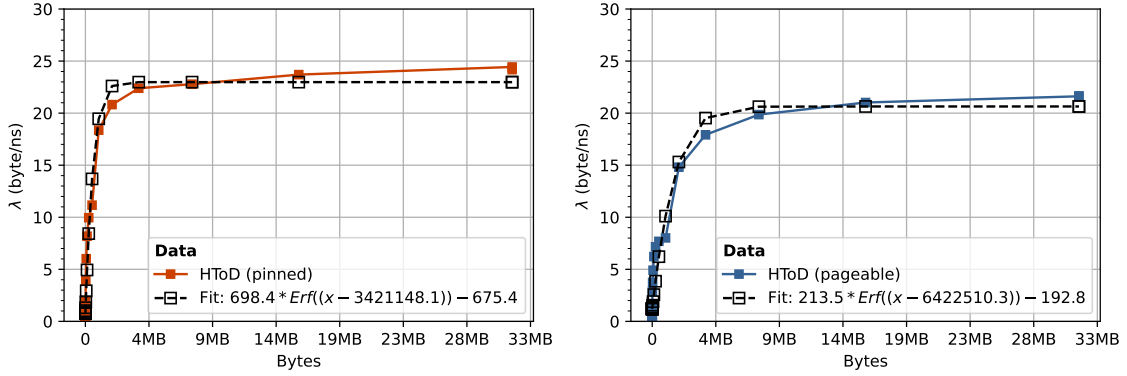


Figure 5.17: Comparison of Gaussian error function fit with microbenchmarked data.

Using our predictors for the throughput, we can model the transfer time for a given data size as follows:

$$\underbrace{2}_{\text{Number of transfers per bulk}} * \underbrace{bulkSize * 8}_{\text{Bulk size in bytes}} * \frac{1}{f_{tp}(bulkSize * 8)}, \quad (5.5)$$

where $f_{tp}(x)$ is the throughput in bytes per second for transferring x bytes. For each bulk of events, we need to transfer two arrays to the GPU: the coordinates and the weights.

To summarise, we model the time spent on transferring a bulk of events in nanoseconds as follows:

Model 5.5: Memory Transfers

$$f_{tp}(d) = \begin{cases} \alpha_{tp} * \text{Erf}((d - \beta_{tp}) * \gamma_{tp}) + \delta_{tp} & d < \tau_4 \\ \max_{tp} & d \geq \tau_4 \end{cases}$$

$$T_{MemCpy_{HToD}}(bulkSize) = 2 * \underbrace{bulkSize * 8}_{\text{Bulk size in bytes}} * \frac{1}{f_{tp}(bulkSize * 8)}$$

Calibration

In our model for the host-to-device memory transfers, we have 6 parameters that need to be calibrated: the transfer size threshold at which the throughput stabilises (τ_4), the four Gaussian error function parameters (α_{tp} , β_{tp} , γ_{tp} , and δ_{tp}) for small transfers, and the maximum throughput for a single transfer (\max_{tp}). These parameters can be calibrated by executing the following steps:

- 1). Microbenchmark the time spent on copying data from the host-to-device with a range of data sizes.
- 2). Plot the throughput in (giga)bytes per second against the transfer size and determine the size at which the throughput stabilises to get the parameter τ_4 .
- 3). To get α_{tp} , β_{tp} , γ_{tp} , and δ_{tp} , fit a function of the form $f_{tp}(d) = \alpha_{tp} * \text{Erf}((d - \beta_{tp}) * \gamma_{tp}) + \delta_{tp}$ with the transfer size in bytes as input d and measured throughput for sizes smaller than τ_4 as output.
- 4). To get \max_{tp} , take the average over the measured throughput for sizes larger τ_4 .

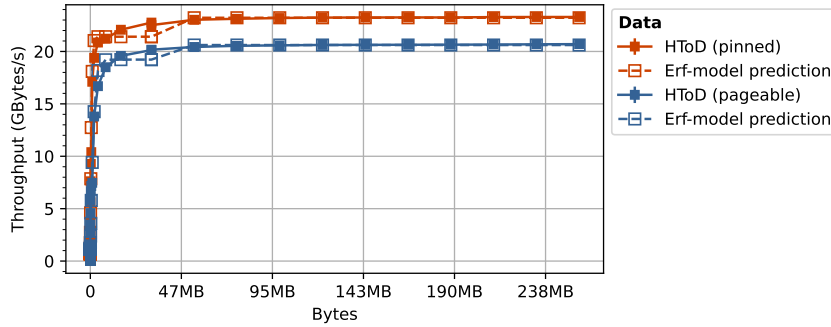


Figure 5.18: Calibration of host-to-device transfer throughput. A polynomial equation was fit to the microbenchmarked data using Python's `scipy.curve_fit`.

5. GPU IMPLEMENTATION

Figure 5.18 shows the result of our fitted Gaussian error function and computed maximum throughput compared to our microbenchmarked data. We used the microbenchmark implementation that is included in the appendix as Listing C.4. The plot confirms that our calibration results in an accurate representation of the microbenchmarked observations. In conclusion, we use the following calibrated values for our model in Model 5.5.

Calibration 5.4: Memory Transfers

- $\tau_4 = 48$ MB

Pinned

- $\alpha_{tp} \approx 698.4$
- $\beta_{tp} \approx -3421148.1$
- $\gamma_{tp} \approx 0$
- $\delta_{tp} \approx -675.4$
- $max_{tp} \approx 24.9$ byte/ns

Pageable

- $\alpha_{tp} \approx 213.5$
- $\beta_{tp} \approx -6422510.3$
- $\gamma_{tp} \approx 0$
- $\delta_{tp} \approx -192.8$
- $max_{tp} \approx 22.1$ byte/ns

Validation

To validate the $T_{MemCpy_{HToD}}$ model, we compare the predicted transfer time with the benchmarked total time spent on host-to-device transfers measured using NSight Systems for different bulk sizes, different histogram sizes, and data distribution. The results of this are displayed in Figure 5.19.

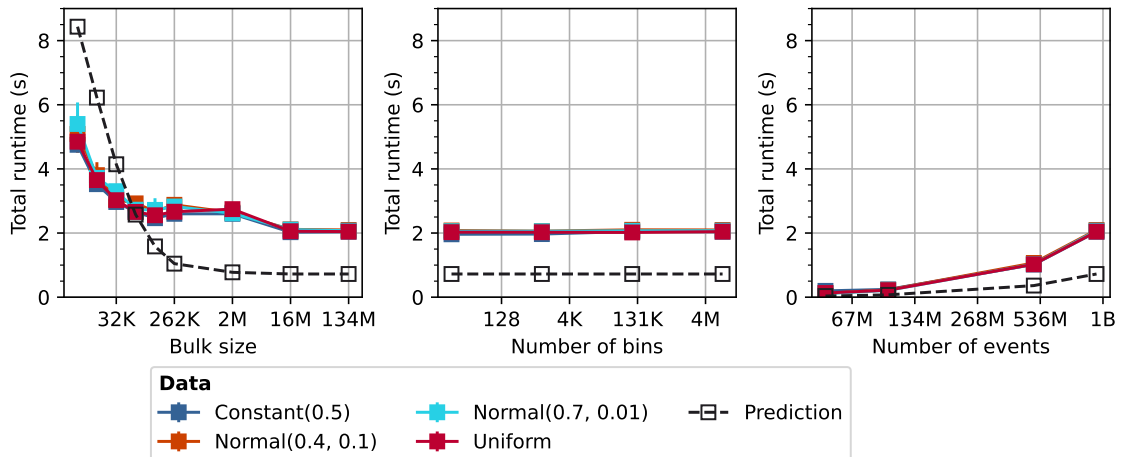


Figure 5.19: Validation of $T_{UpdateStats_{GPU}}$. Average runtime taken over 5 runs for different bulk sizes, number of bins, and number of events. In each subplot, we fix two of the three parameters to the maximum tested value.

The figure shows that our model can accurately predict the time spent on host-to-device transfers, with a deviation of at most two seconds at 1 billion events. For small bulk sizes, the model overestimates the transfer time, while for larger sizes, it underestimates the transfer time. This could indicate that our model is missing another aspect that is influenced by the bulk size, aside from the PCIe bandwidth. Since the absolute prediction error remains relatively small, we expect this model to be sufficient for predicting relevant performance benefits.

5.5 Total Model Validation

To validate our complete model for the GPU, we run the same benchmark and use the same parameters as for our validation of the CPU model. Figure 5.20 compares the measured runtimes for different bulk sizes, histogram sizes, number of events, type of bins, and data distributions, against our model predictions.

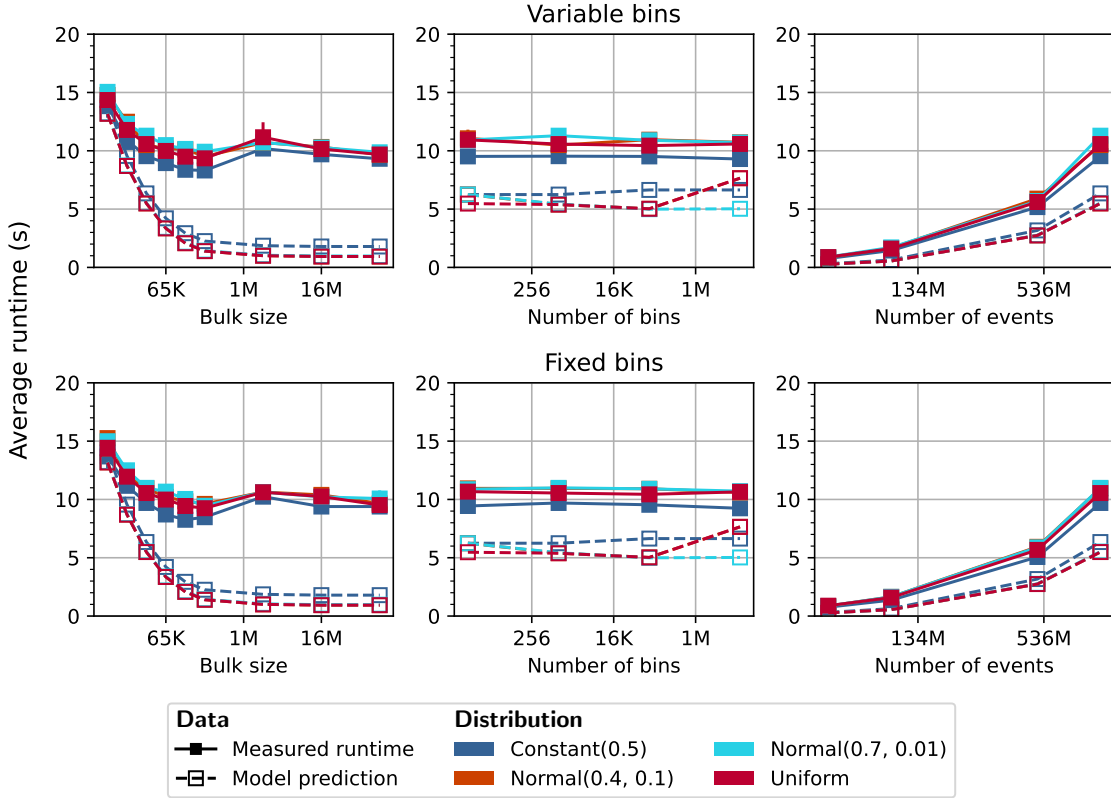


Figure 5.20: Validation of $T_{Total_{GPU}}$. Measured runtimes are plotted as the average over 5 runs with standard deviation error bars. The three plots on the top row contain results for histograms with variable bins, and the bottom row for fixed bins. In each plot, we vary the parameter labelled on the x-axis and fix the other parameters to the maximum value tested. For example, the plot on the top-left displays the result for different bulk sizes 10M variable-sized bins and 1B events in total.

Focusing on each parameter individually, we observe the following:

- **Bulk size:** the benchmark results show that the runtime decreases for larger bulk sizes up to 256K and then stabilises. Our model accurately captures this trend.
- **Type of bins:** there is no significant difference in the measured runtime performance between variable and fixed bins. This validates our assumption that it is sufficient to only model the most expensive case (variable bins).

- **Number of bins:** according to the measured runtimes, the histogram size has a minimal impact on the performance, unlike on the CPU. While our model also showcases this, there is one point with a significant jump: 100M bins with a uniform distribution. This is largely due to an estimated increase in cache misses in the binary search. This effect may not have been observed in our benchmarks, because we did not use a sufficiently large histogram.
- **Data distribution:** Based on our benchmark results, the runtime performance for the different distributions can be ranked from best to worst as: *Constant* \rightarrow *Uniform* \rightarrow *Normal*(0.4, 0.1) \rightarrow *Normal*(0.7, 0.01). However, with our model, the ranking changes for different histogram sizes. Although, the distribution has a minimal impact on the runtime compared to the CPU. Therefore, for speedup comparisons, we could use only the predictions for constant distributions, which is the most accurate.
- **Number of events:** similarly to the CPU, the runtime scales linearly with the number of events, which validates our approach to predict the performance for n events by multiplying our model output for a bulk size of b by $\frac{n}{b}$. Since our model underestimates the runtime for a single bulk, the error is magnified for a large number of events.

Overall, our model underestimates the runtime performance, but it does accurately capture the trend. On the other hand, in the validation of the individual components, we observed a relatively high accuracy for the runtime predictions. This indicates that our model could be missing additional interactions when combining components, such as an increase in register pressure. While the performance ranking for different data distributions and histogram sizes is not preserved, the absolute difference in runtimes between the ranking entries is relatively low and thus acceptable.

6

Scenario Analyses

In this chapter, we use our models to perform scenario analyses. More specifically, given different input values for our parameters – number of events, bulk size, histogram size, type of bins, and data distribution – we aim to quantify the speedup of GPU execution compared to execution on only the CPU. Additionally, we use the models to determine when overlapping communication with computation becomes feasible, to further optimise the runtime performance of our current GPU implementation. First, we give an overview of our models.

6.1 Overview of Performance Models

To summarise, we designed the following models for the CPU and GPU execution of the RDataFrame histogramming action:

CPU Model

$$T_{Total_{CPU}} = \underbrace{\left\lceil \frac{nEvents}{bulkSize} \right\rceil}_{\text{Number of bulks}} * \left(T_{LoadBulk_{CPU}} + bulkSize * (T_{FindBin_{CPU}} + T_{AddBinContent_{CPU}} + T_{UpdateStats_{CPU}}) \right)$$

Performance models for runtime per event:

$$f_{subBins}(distr, nBins) = \begin{cases} 1 & \text{distr is Constant} \\ \text{bins_in_range}(\mu - 3\sigma, \mu + 3\sigma) & \text{distr is Normal}(\mu, \sigma) \\ nBins & \text{distr is Uniform} \end{cases}$$

$$f_{bscm}(n, m, C) = \begin{cases} \max\left(\log_2(m) - \log_2(C - \log_2\left(\frac{n}{m}\right)), 0\right) & C > \log_2\left(\frac{n}{m}\right) \\ \log_2(n) - C & \text{Otherwise} \end{cases}$$

$$\begin{aligned} T_{BinarySearch}(nBins, distr) &= \log_2(s) * T_{L1} + f_{bscm}(nBins, s, L1_size) * T_{L2} \\ &+ f_{bscm}(nBins, s, L2_size) * T_{L3} \\ &+ f_{bscm}(nBins, s, L3_size) * T_{mmem}, \\ s &= f_{subBins}(distr, nBins) \end{aligned}$$

$$T_{FindBinCPU}(binType, nBins, distr) = \begin{cases} T_{Algorithm\ 1:line\ 10} & \text{Fixed bins} \\ T_{BinarySearch}(nBins, distr) & \text{Variable bins} \end{cases}$$

$$\begin{aligned} T_{AddBinContentCPU}(nBins, distr) &= \\ &T_{L1} + f_{abcm}(s, L1_size) * T_{L2} \\ &+ f_{abcm}(s, L1_size) * f_{abcm}(s, L2_size) * T_{L3} \\ &+ f_{abcm}(s, L1_size) * f_{abcm}(s, L2_size) \\ &* f_{abcm}(s, L3_size) * T_{mmem}, \\ s &= f_{subBins}(distr, nBins) \end{aligned}$$

$$T_{UpdateStatsCPU} = T_{Alg3-3:7}$$

GPU Model

$$T_{TotalGPU} = \underbrace{\left\lceil \frac{nEvents}{bulkSize} \right\rceil}_{\text{Number of bulks}} * \left(T_{LoadBulkCPU} + T_{HtoDBulk} + T_{FindBinGPU} + T_{AddBinContentGPU} + T_{UpdateStatsGPU} \right)$$

Performance models for runtime per bulk:

6. SCENARIO ANALYSES

$$f_{bsl}(n) = \begin{cases} \lambda_{bsl1} & n < \tau_1 \\ \lambda_{bsl2} & n < \tau_2, \tau_2 > \tau_1 \\ \lambda_{bsl3} & n \geq \tau_2, \tau_2 > \tau_1 \end{cases}$$

$$T_{FindBinGPU}(bulkSize, distr, nBins) = \frac{1}{t_{fb}} * bulkSize * f_{bsl}(f_{subBins}(distr, nBins))$$

$$T_{atomic}(nBins, x) = \begin{cases} \alpha_{ag}x + \beta_{ag} & nBins > sharedMemory \text{ (Global)} \\ \alpha_{al}x + \beta_{al} & nBins \leq sharedMemory \text{ (Local)} \end{cases}$$

$$T_{AddBinContentGPU}(bulkSize, distr, nBins) = T_{atomic}\left(nBins, \frac{bulkSize}{f_{subBins}(distr, nBins)}\right)$$

$$T_{transformReduce}(nThreads) = \alpha_{tr} + nThreads + \beta_{tr}$$

$$T_{UpdateStatsGPU}(bulkSize) = \underbrace{4}_{\text{Number of statistics}} * \left(T_{transformReduce}(bulkSize) + T_{transformReduce}\left(\underbrace{\left\lceil \frac{bulkSize}{2 * 256} \right\rceil}_{\text{Number of CUDA blocks}}\right) \right)$$

$$f_{tp}(d) = \begin{cases} \alpha_{tp} * Erf((d - \beta_{tp}) * \gamma_{tp}) + \delta_{tp} & d < \tau_4 \\ \max_{tp} & d \geq \tau_4 \end{cases}$$

$$T_{MemCpyHToD}(bulkSize) = 2 * \underbrace{bulkSize * 8}_{\text{Bulk size in bytes}} * \frac{1}{f_{tp}(bulkSize * 8)}$$

Notes:

- Parameters that need to be calibrated to the system are **highlighted**.
- `bins_in_range(a,b)` indicates the number of bins that are allocated to the coordinate range [a,b] based on the fixed bin width or specified bin edges.
- $T_{AlgA-B:C}$ denotes the execution time of code lines B to C in Algorithm A.

6.2 Speedup

In this section, we use our models to predict the performance benefit of moving the histogram computation to the GPU. We compute the speedup as follows:

$$Speedup = \frac{T_{CPU}}{T_{GPU}}, \quad (6.1)$$

where T_{CPU} is the runtime of the histogram action on the CPU with a single thread, and T_{GPU} is the runtime of the GPU without asynchronous execution. To predict the speedup of the GPU against CPU execution with p threads, an upper bound estimation can be made by dividing T_{CPU} by p in Equation 6.1. This gives an upper bound because it assumes perfect overlap of events processing between different threads and no additional computations. In reality, parallelising bulks of events requires synchronisation or an additional post-processing step to combine histogram results.

To verify the speedup predictions, we compare the output of our models with speedups computed using the measured runtimes in the validation of the CPU and GPU models. The results of this comparison are displayed in Figure 6.1

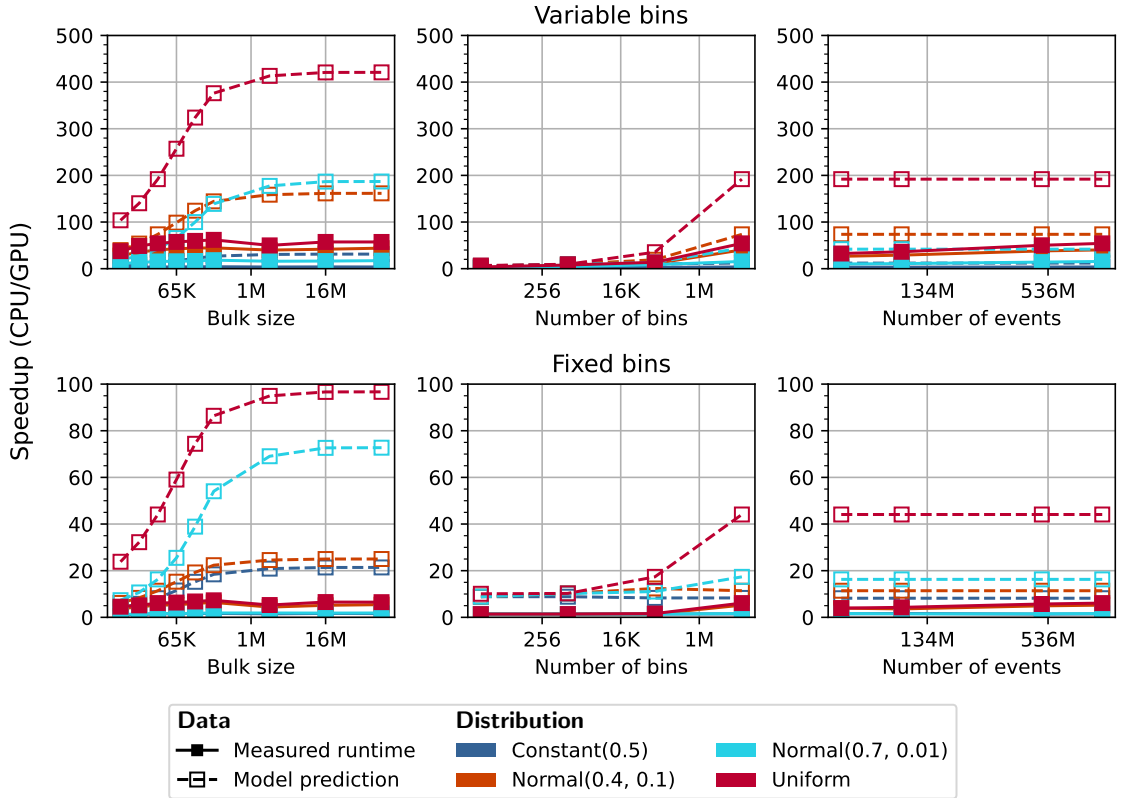


Figure 6.1: Predicted and measured speedups. Predicted speedups are calculated with $T_{TotalCPU}$ and $T_{TotalGPU}$. Measured speedups use the data from Figures 4.12 and 5.20.

6. SCENARIO ANALYSES

The results displayed in Figure 6.1 show that our models considerably overestimate the speedup. As seen in the previous validations section 4.4 and 5.5, our CPU performance model is optimistic (underestimates) about the runtime, while our GPU model is pessimistic (overestimates). This combination results in exaggerated speedups, which implies that our current model is unsuitable for absolute speedup analyses. Since we cannot make accurate predictions with our current model for single-threaded CPU execution, we do not proceed with a comparison against multi-threaded execution.

6.3 Overlapping Computation with Communication

In the current GPU implementation, we execute all GPU operations in a single CUDA stream, i.e., we do not overlap communication with computation. As a result, before the processing of each bulk on the GPU, we have a section where the GPU is idle because it is waiting for the event data to be transferred into the device memory. An obvious optimisation to consider is therefore overlapping the transfer overhead of a bulk with the execution of a kernel.

One implementation option is to mask the memory copying overhead by transferring the data of the next bulk while the GPU is processing the current bulk. To perfectly hide the transfer latency, the total kernel runtime for bulk needs to be greater than or equal to the time spent on copying. Based on our GPU performance model, this would be when the following holds:

$$\begin{aligned}
 &T_{FindBin_{GPU}}(bulkSize, distr, nBins) \\
 &+ T_{AddBinContent_{GPU}}(bulkSize, distr, nBins) \\
 &+ T_{UpdateStats_{GPU}}(bulkSize) \geq T_{MemCpy_{HToD}}(bulkSize)
 \end{aligned} \tag{6.2}$$

In this comparison, we have three parameters: the bulk size, the data distribution, and the number of bins. The data distribution affects how many bins are accessed, so we can simplify the comparison by combining the last two parameters. In Figure 6.2, we plot the expected memory transfer overhead and total kernel runtime for different bulk sizes and the number of accessed bins. Since we have two different kernels for filling the bins in the "Add Bin Content" component, one with an intermediate filling of a local histogram copy (Local) and one where we fill the global histogram directly (Global), we plot the predictions separately for the two versions.

6.3 Overlapping Computation with Communication

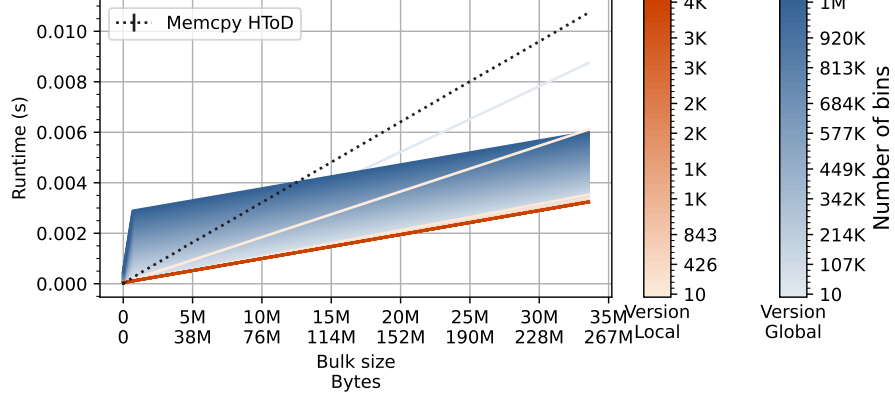


Figure 6.2: Predicted kernel runtimes and memory transfer overhead. The Global and Local versions refer to the type of kernel used in “Add Bin Content”

Based on Figure 6.2, version Local always executes faster than the memory copy, while with the Global version and 1M bins, the kernel runtime for a single bulk longer than the transfer time for bulk sizes up to $\sim 13M$. On the other hand, it is unclear how common histograms with 1 million bins are. Furthermore, we notice that the kernel runtime up to a bulk size of $\sim 13M$ is still only around 0.004 seconds. Also, the processing of two bulks of size $\sim 13M$ with overlapping seems to only result in a slight improvement compared to simply computing a bulk of $\sim 26M$ without overlap (3×0.004 vs. $0.005 + 0.008$). With the addition of the overhead of creating multiple CUDA streams, the benefit of overlapping in this case is therefore likely minimal.

From this, we can see that the execution of a single histogram action is too memory-bound to be able to increase efficiency by overlapping communication with computation. To remedy this, we either need more histogram computations that (partially) utilise data from the same bulks, or we need other GPU operations to increase the kernel runtime.

Related Work

To the best of our knowledge, this is the first attempt at modelling the performance of ROOT computational actions within RDataFrame. However, other analytical performance modelling studies exist that focus on different aspects of a GPU application: the kernel runtime, memory transfer overheads, and models that combine these two. We highlight a few of these works that have a similar objective, in no particular order.

Kernel execution models

Sitchinava and Weichert propose a basic theoretical model for modelling the upper bound of the total runtime of an application [38]. The model considers the number of parallel memory blocks transferred between the shared memory and global memory, the number of time units needed to execute the kernel instructions, and the number of global synchronisation.

Koike and Sadakane propose a computational model for analysing the asymptotic behaviour of an algorithm based on the number of executed instructions, the number of issued global memory access instructions, and the amount of shared and global memory used [24]. For the time complexity, the model takes into account branch divergence and hardware multi-threading.

Zhang and Owens derive a throughput model from micro-benchmarks of the instruction pipeline, shared memory, and global memory patterns [41]. By inspecting the native GPU code and simulating the memory transactions, the model can predict the expected instruction throughput and memory bandwidth.

Memory transfer models

Werkhoven et al. propose an analytical performance model based on the LogGP model [2] that includes PCIe transfers and overlapping computation with communication using

either device-mapped memory or different streams. The total execution time is modelled as a linear combination of the transfer times from host to device, the kernel execution time, and the transfer times from device to host, parameterised by the number of streams. They make a distinction between cases with and without implicit synchronisation, where the kernel or transfers are dominant, and with one or two copy engines. In addition, they extended the classic roof-line model for applications with communication-computation overlaps. However, modelling the kernel execution time is outside of the paper’s scope.

Riahi et al. compare three machine-learning-based methods (regression, Random Forest regression, and Neural Networks) and a modified GROPHECY++ model [6], the λ -Model, for predicting the data transfer time between the CPU and GPU through the PCIe bus. They developed micro-benchmarks to train the ML models and to determine the values for the parameters in the analytical model. In their experiments, the results show that the bandwidth is variable for small data transfers up to a certain threshold, where the bandwidth remains stable because the maximum is hit. Therefore, they recommend using a hybrid model: the λ -Model for large-sized data and regression methods for small-sized data.

Combination models

Gómez-Luna et al. present mathematical models for estimating the execution time of a data-parallel application where the data is broken up into several chunks to overlap computation with asynchronous communication using multiple streams for a staged execution [16].

Carroll and Wong propose an analytical model, ATGPU, for modelling the total execution time of a GPU algorithm [9]. The model combines the work of Sitchinava and Weichert [38] and Koike and Sadakane [24], to develop a model that captures both the kernel execution and the data transfer time in the prediction. ATGPU analyses algorithms in *rounds*, where a round begins with a transfer of data from the host to device’s global memory. This is followed by the execution of a kernel and ends with a transfer of output data from the global memory to the host. Within the kernel, the model considers branching and shared memory access patterns.

Generally, existing works describe modelling approaches for generic applications and are tested only with mini-apps. In contrast, our work models both kernel execution time and memory transfer overheads, leveraging application-specific knowledge. Furthermore, we validate our approach within a large and complex codebase, demonstrating its applicability to real-world scenarios.

Conclusion

To conclude, we summarise our main findings, provide an answer to our research question, we outline the limitations of our work, and provide suggestions for future research.

8.1 Summary and Main Findings

We proposed a systematic approach for designing an analytical runtime performance, which involves 1). identifying the components dependent on different parameters. 2). investigating the factors influencing its runtime via code analysis, algorithm analysis, and/or microbenchmarking, 3). designing a mathematical equation based on the observations. 4). calibrating architecture-dependent parameters, 5). and validating the model against measured runtimes to evaluate the accuracy.

We demonstrated this strategy for a CPU and GPU implementation of RDataFrame histogramming to predict the performance benefits of moving computations from the CPU to the GPU. In both implementations, we iterate over the event data in bulks to fill a histogram. Our model can predict the runtime performance for various parameters such as bulk sizes, histogram sizes, bin types (fixed-size or variable-sized), total number of events, and data distributions. Using short-running microbenchmarks, we adapt the model to different system specifications.

While our CPU model generally overestimates the absolute runtime, and the GPU model underestimates, both accurately capture trends and performance rankings for most parameters. On the other hand, the validation of the individual components indicated a high accuracy for the runtime predictions. This suggests that the total runtime is not simply a sum of our components' runtimes and that our model is missing more complicated interactions between the components. Another factor that our model neglects, is potential data sorting (i.e., different distributions per bulk), which can influence the memory latencies.

While the accuracy of the models can be further improved by capturing potential dependencies between components or other aspects such as changing data distributions per bulk, this will add more complexity to the model. In addition, more data needs to be collected to define the model, which can be time-consuming. Before adding another layer to a performance model, consider this trade-off between effort and accuracy. Due to the inaccuracies in our models, the acquired speedup predictions were unrealistically high. However, the model's predictions and the design process helped us gain a better understanding of the implementation's performance bottlenecks.

Coming back on our research question:

How can we predict the runtime performance benefits of offloading computations to the GPU?

Based on our findings, we can conclude that an analytical model that considers the number of executed instructions, cache performance, and atomic contention is suitable for trend analysis and performance optimisation discovery. For accurate runtime predictions, a more complex model is necessary, albeit requiring more effort to design, use, and interpret.

8.2 Limitations

Despite our comprehensive analysis, results may vary in different setups due to limitations. Firstly, our models' accuracy may change on different architectures, as calibration methods were not analysed with other CPUs and GPUs. Secondly, while our base models can be easily modified to predict performance for higher levels of parallelism, accuracy for these scenarios is unverified.

8.3 Future work

To address the aforementioned limitations, we suggest testing the models on different architectures and extending them for higher levels of parallelism such as multi-threaded, distributed, and pipelined processing of event bulks.

Moreover, it is worth investigating how offloading more RDataFrame actions and executing a complete workflow on the GPU can enhance performance. Since this work was the first venture into GPU support within RDataFrame, many other RDataFrame actions remain to be investigated. Porting more RDataFrame actions allows for larger computational workloads on the GPU, which is necessary to improve the performance of our memory-bound implementations.

References

- [1] Nasim Ahmed, Andre LC Barczak, Mohammad A Rashid, and Teo Susnjak. 2022. Runtime prediction of big data jobs: performance comparison of machine learning algorithms and analytical models. *Journal of Big Data* 9, 1 (2022), 67. doi:10.1186/s40537-022-00623-1 3
- [2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1995. LogGP: Incorporating Long Messages into the LogP Model—one Step Closer towards a Realistic Model for Parallel Computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures* (Santa Barbara, California, USA) (*SPAA '95*). Association for Computing Machinery, New York, NY, USA, 95–105. doi:10.1145/215399.215427 72
- [3] AMD. 2023. *AMD EPYC™ 7402P*. AMD. <https://www.amd.com/en/products/cpu/amd-epyc-7402p> Accessed on 13-12-2023. 82
- [4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15* (Delft, The Netherlands). Springer, Springer, Berlin, Heidelberg, 863–874. doi:10.1007/978-3-642-03869-3_80 4
- [5] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer* 49, 5 (2016), 54–63. doi:10.1109/MC.2016.127 9, 82, 83
- [6] Michael Boyer, Jiayuan Meng, and Kalyan Kumaran. 2013. Improving GPU Performance Prediction with Data Transfer Modeling. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (Cambridge,

REFERENCES

- MA, USA). IEEE, New York City, USA, 1097–1106. doi:10.1109/IPDPSW.2013.23673
- [7] Rene Brun and Fons Rademakers. 1997. ROOT — An object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 389, 1 (1997), 81–86. doi:10.1016/S0168-9002(97)00048-X New Computing Techniques in Physics Research V. 1
- [8] C Sidney Burrus, James W Fox, Gary A Sitton, and Sven Treitel. 2003. Horner’s method for evaluating and deflating polynomials. *DSP Software Notes, Rice University, Nov 26* (2003), 1–9. <https://web.archive.org/web/20240326132436/https://www-ece.rice.edu/dsp/software/FVHDP/horner2.pdf> 15
- [9] Thomas C. Carroll and Prudence W.H. Wong. 2017. An Improved Abstract GPU Model with Data Transfer. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)* (Bristol, UK). IEEE, New York City, USA, 113–120. doi:10.1109/ICPPW.2017.28 73
- [10] Jolly Chen. 2023. *ROOT CUDA Histogramming development branch*. ROOT. https://github.com/jolly-chen/root/tree/gpu_histogram_bulk Latest commit: <https://github.com/jolly-chen/root/commit/7bc23323b503cdd7b3dafbf1792ca599b7048fed>. 9, 83
- [11] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 28)*, Sanjoy Dasgupta and David McAllester (Eds.). PMLR, Atlanta, Georgia, USA, 1337–1345. <https://proceedings.mlr.press/v28/coates13.html> 2
- [12] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters* 21, 02 (2011), 173–193. doi:10.1142/S0129626411000151 4
- [13] Google. 2023. *Google Benchmark*. Google. <https://github.com/google/benchmark> Latest commit: <https://github.com/google/benchmark/commit/159eb2d0ffb85b86e00ec1f983d72e72009ec387>. 16

REFERENCES

- [14] Enrico Guiraud. 2023. *ROOT RDataFrame bulk processing development branch*. ROOT. <https://github.com/eguiraud/root/tree/df-bulk-ntuple> Latest commit: <https://github.com/eguiraud/root/commit/3cb95f7b1b321a2e24329ce8bdb88729f235ae54>. 6
- [15] Enrico Guiraud, Jakob Blomer, Philippe Canal, and Axel Naumann. 2023. Boosting RDataFrame performance with transparent bulk event processing. In *26th International Conference on Computing in High Energy & Nuclear Physics (CHEP2023)* (Norfolk Waterside Marriott). CHEP, IEEE, New York City, USA, preprint. <https://indico.jlab.org/event/459/contributions/11565/> 6
- [16] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. 2012. Performance models for asynchronous data transfers on consumer Graphics Processing Units. *J. Parallel and Distrib. Comput.* 72, 9 (2012), 1117–1126. doi:10.1016/j.jpdc.2011.07.011 Accelerators for High-Performance Computing. 73
- [17] Torsten Hoefer, Timo Schneider, and Andrew Lumsdaine. 2010. LogGOPSim: simulating large-scale applications in the LogGOPS model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 597–604. doi:10.1145/1851476.1851564 3
- [18] S. Huang, S. Xiao, and W. Feng. 2009. On the energy efficiency of graphics processing units for scientific computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing* (Rome, Italy). IEEE, New York City, USA, 1–8. doi:10.1109/IPDPS.2009.5160980 2
- [19] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. 2014. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence* 206 (2014), 79–111. doi:10.1016/j.artint.2013.10.003 3
- [20] Intel. 2024. *Intel VTune Profiler: Find and Fix Performance Bottlenecks Quickly and Realize All the Value of Your Hardware*. Intel. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html> Accessed on 26-04-2024. 4
- [21] Awais Khan, Hyogi Sim, Sudharshan S. Vazhkudai, Ali R. Butt, and Youngjae Kim. 2021. An Analysis of System Balance and Architectural Trends Based on Top500 Supercomputers. In *The International Conference on High Performance Computing*

REFERENCES

- in Asia-Pacific Region* (Virtual Event) (*HPCAsia '21*). Association for Computing Machinery, New York, NY, USA, 11–22. doi:10.1145/3432261.3432263 1
- [22] Paul-Virak Khuong and Pat Morin. 2017. Array layouts for comparison-based searching. *Journal of Experimental Algorithmics (JEA)* 22 (2017), 1–39. doi:10.1145/3053370 20, 21
- [23] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S Vetter. 2021. IRIS: A portable runtime system exploiting multiple heterogeneous programming systems. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)* (Waltham, MA, USA). IEEE, IEEE, New York City, USA, 1–8. doi:10.1109/HPEC49654.2021.9622873 4
- [24] Atsushi Koike and Kunihiro Sadakane. 2015. A Novel Computational Model for GPUs with Applications to Efficient Algorithms. *International Journal of Networking and Computing* 5, 1 (2015), 26–60. doi:10.15803/ijnc.5.1_26 72, 73
- [25] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (San Diego, CA) (*ATEC '96*). USENIX Association, USA, 23. <https://lmbench.sourceforge.net/lmbench-usenix.pdf> 25
- [26] NVIDIA. 2024. *CUDA C++ Programming Guide*. NVIDIA. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/> Accessed on 30-04-2024. 6
- [27] NVIDIA. 2024. *NVIDIA Nsight Compute*. NVIDIA. <https://developer.nvidia.com/nsight-compute> Accessed on 26-04-2024. 4
- [28] Padulano, Vincenzo Eduardo, Cervantes Villanueva, Javier, Guiraud, Enrico, and Tejedor Saavedra, Enric. 2020. Distributed data analysis with ROOT RDataFrame. *EPJ Web Conf.* 245 (2020), 03009. doi:10.1051/epjconf/202024503009 1, 8
- [29] Danilo Piparo, Philippe Canal, Enrico Guiraud, Xavier Pla, Gerardo Ganis, Guilherme Amadio, Axel Naumann, and Enric Tejedor. 2019. RDataFrame: Easy Parallel ROOT Analysis at 100 Threads. *EPJ Web of Conferences* 214 (01 2019), 06029. doi:10.1051/epjconf/201921406029 1, 4, 8
- [30] Friedrich Pukelsheim. 1994. The three sigma rule. *The American Statistician* 48, 2 (1994), 88–91. doi:10.1080/00031305.1994.10476030 23

REFERENCES

- [31] Ali Riahi, Abdorreza Savadi, and Mahmoud Naghibzadeh. 2020. Comparison of analytical and ML-based models for predicting CPU–GPU data transfer time. *Computing* 102, 9 (2020), 2099–2116. doi:10.1007/s00607-019-00780-x 58, 73
- [32] Thomas Roehl. 2022. *likwid-pin: Tool to pin threaded applications without touching the source code*. RRZE-HPC. <https://github.com/RRZE-HPC/likwid/wiki/Likwid-Pin> Revision 8. 87
- [33] ROOT. 2023. *Building ROOT from source*. ROOT. https://root.cern/install/build_from_source/ Accessed on 06-12-2023. 83
- [34] ROOT. 2023. *Histogram Library*. ROOT. https://root.cern/doc/v630/group_Hist.html 13
- [35] ROOT. 2023. *ROOT::RDataFrame Class Reference*. ROOT. https://root.cern/doc/master/classROOT_1_1RDataFrame.html 4
- [36] ROOT. 2024. *ROOT::Experimental::RNTuple Class Reference*. ROOT. https://root.cern/doc/master/classROOT_1_1Experimental_1_1RNTuple.html Accessed on 29-04-2024. 11
- [37] R. Rugina and K.E. Schauser. 1998. Predicting the running times of parallel programs by simulation. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. 654–660. doi:10.1109/IPPS.1998.669996 3
- [38] Nodari Sitchinava and Volker Weichert. 2013. Provably Efficient GPU Algorithms. *ArXiv* abs/1306.5076 (2013). <https://api.semanticscholar.org/CorpusID:14849470> 72, 73
- [39] The ATLAS Collaboration. 2012. Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. *Physics Letters B* 716, 1 (2012), 1–29. doi:10.1016/j.physletb.2012.08.020 5
- [40] B. van Werkhoven, J. Maassen, F.J. Seinstra, and H.E. Bal. 2014. Performance Models for CPU-GPU Data Transfers. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (Chicago, IL, USA). IEEE, New York City, USA, 11–20. doi:10.1109/CCGrid.2014.16 72
- [41] Yao Zhang and John D. Owens. 2011. A quantitative performance analysis model for GPU architectures. In *2011 IEEE 17th International Symposium on High Performance*

REFERENCES

Computer Architecture (San Antonio, TX, USA). IEEE, New York City, USA, 382–393. doi:10.1109/HPCA.2011.5749745 72

Appendix A

Experimental Setup

A.1 Test Systems

CPU Specifications	DAS-6 [5]
Processor	AMD EPYC 7402P [3]
Sockets	1
Cores per socket	24
Threads per core	2
Base clock speed	2.80 GHz
Max boost clock speed	3.35 GHz
L1D cache size	32 KB
L1D cache line size	64 B
L1D cache associativity	8
L2 cache size	512 KB
L2 cache line size	64 B
L2 cache associativity	8
L3 cache size	16 MB
L3 cache line size	64 B
L3 cache associativity	0
Main memory size	128 GB
Memory channels	8
Memory bandwidth	204.8 GB/s

Table A.1: Description of the CPU used for benchmarking and validation.

GPU Specifications	DAS-6 [5]
Processor	NVIDIA RTX A4000
Compute Capability	8.6
Architecture	Ampere
Driver versions	545.23.06
CUDA Cores	6144
SMs	48
Shared memory	48 KB
Global memory	16 GB
Memory clock	6836 MHz
Memory bus width	256 bits
Memory bandwidth	448.1 GB/s

Table A.2: Description of the GPU used for benchmarking and validation.

A.2 ROOT Setup

In this section, we describe the steps required to build the GPU histogramming development branch of ROOT. The implementation is available in a fork of ROOT, at [10]. This branch is based on the ROOT version 6.31/01. For the minimum requirements and the dependencies, see the official ROOT installation page [33]. After cloning the repository, the following command needs to be executed to build the project:

Listing A.1: CMake build command

```
cmake ../root/ -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_STANDARD=17
--Dcuda=ON -DCMAKE_CUDA_ARCHITECTURES=<cuda_arch>
--DCUDA_TOOLKIT_ROOT_DIR=</path/to/cuda-toolkit>
--DCMAKE_CUDA_COMPILER=<path/to/nvcc>
```

Once ROOT has been built, the command `source bin/thisroot.sh` needs to be run to add the ROOT libraries to the environment.

Appendix B

Microbenchmarking

B.1 Measuring Runtimes with Google Benchmark

To determine the best method for measuring runtimes using Google Benchmark’s framework, we created several benchmarks that measure known durations to determine their accuracy. The basic structure of a benchmark in this framework is shown in Listing B.1. Each benchmark consists of a main work-loop `for (auto _ : state)`, that repeats the execution of a given workload. Global setup operations are placed outside the loop, while initialization operations that need to be performed every iteration are placed inside the loop. However, by default, the framework measures the entire duration of the work-loop.

Listing B.1: Google Benchmark structure

```
1 static void BM_Basic(benchmark::State &state) {  
2     // Global setup here  
3     for (auto _ : state) {  
4         // Setup per iteration  
5         // Workload  
6     }  
7     // Teardown  
8 }
```

There are two methods to exclude the per-iteration setup operations from the timing results:

- **Pause timer:** we can pause the timer before the setup operations and resume the timer afterwards.

```
1 static void BM_PauseOverhead(benchmark::State &state) {  
2     for (auto _ : state) {  
3         state.PauseTiming();  
4         // Some setup
```


B.1 Measuring Runtimes with Google Benchmark

```
5         state.ResumeTiming();
6         std::this_thread::sleep_for(std::chrono::milliseconds(1));
7     }
8 }
9 BENCHMARK(BM_PauseOverhead)->Unit(benchmark::kMillisecond);
```

- **Manual timer:** we manually time the workload and set the iteration time explicitly.

```
1 static void BM_ManualOverhead(benchmark::State &state) {
2     for (auto _ : state) {
3         // Some setup
4
5         auto start = std::chrono::steady_clock::now();
6         std::this_thread::sleep_for(std::chrono::milliseconds(1));
7         auto end = std::chrono::steady_clock::now();
8
9         auto elapsed_seconds = std::chrono::duration_cast<
10             std::chrono::duration<double>>(end - start);
11         state.SetIterationTime(elapsed_seconds.count());
12     }
13 }
14 BENCHMARK(BM_ManualOverhead)
15     ->UseManualTime()
16     ->Unit(benchmark::kMillisecond);
```

To compare the accuracy between the two different timing options and to confirm that these exclude the time spent in setup operations, we measured different durations of `std::this_thread::sleep_for`, which blocks the execution of a thread for a given duration. As a mock setup action, we initialize a vector of 10 million elements with random values.

Sleep duration	Pause Timer	Manual Timer
1 Nanosecond	51925 ns	51911 ns
1 Microsecond	52.8 us	52.7 us
1 Millisecond	1.10 ms	1.10 ms
1 Second	1 s	1 s

Table B.1: Comparison between the two different timing options.

The average results over three repetitions for the different timing methods are displayed in Table B.1, which shows that there is a negligible difference between the two methods. Additionally, we observe that the measured time deviates considerably from the expected value for units smaller than a millisecond. However, this is more likely due to overhead

within the sleep operation than the timer being inaccurate. While we did not notice a significant difference in the measured time between the two options, we decided to use the method with a manual timer, since the user guide discourages pausing the timer.

B.2 Microbenchmark Guidelines

A microbenchmark is a benchmark designed for measuring the performance of a small operation, in isolation. There are many implementation aspects that can cause the microbenchmark to perform differently, when compared to the execution of the small operation integrated within a larger program. These need to be taken into consideration when using a microbenchmark to evaluate the performance of a procedure. We highlight four aspects, ordered from high to low impact, and provide suggestions to mitigate the impact when using a microbenchmark to measure runtime and cache performance:

- 1). **Compiler optimisations:** due to compiler optimisations, the microbenchmark snippet can perform better than in reality. For example, unused variables can be eliminated or operations involving constants can be simplified by the compiler, which results in fewer instructions being executed. If the variables are not actually unused or constants in the full program, the microbenchmark would execute less code than intended.

Mitigation: inspect the compiled assembly code, to verify that the procedure executes the expected instructions. A useful tool for this is GodBolt, which displays the generated assembly code when using different compilers and compiler flags. A quick way to determine that a code line is not optimised away, is to check that the line is highlighted in the source code view, which indicates that there is assembly code linked to it. Furthermore, by right-clicking on the source code line and selecting “*Reveal linked code*”, the assembly code view jumps to the corresponding line, where the type of instructions can be evaluated.

- 2). **Number of iterations:** by definition, the computations in a microbenchmark are very short, which cannot be captured with sufficient accuracy due to the limited resolution of the clock used to measure intervals.

Mitigation: repeat the computation several times and take the average to get the runtime for a single instance. The added loop, however, can result in some artificial overhead in the obtained measurements. This can be minimized by doing multiple repetitions per loop-iteration (i.e., unrolling the loop).

- 3). **Data alignment:** depending on how a vector of data is aligned in the memory,

consecutive elements may end up in different cache lines. For example, given a cache line size of 8 doubles, the first 8 doubles of a vector could be mapped to two different lines – even though one line is sufficient – if the start of the vector is not aligned with a cache line. As a result, the microbenchmark could produce unexpected cache miss-ratios.

Mitigation: use an allocator that ensures alignment.

- 4). **Thread migration:** when running an application, the operating system scheduler can occasionally move the program thread to a different core, to balance the load on the available processors. In addition to the migration overhead, this can also lead to an increased number of cache misses, as the higher-level caches are typically per-core.

Mitigation: pin the threads to specific cores with a tool such as `likwid-pin` [32].

Appendix C

GPU Calibration Microbenchmarks

The full benchmark suite for calibrating our models can also be found on GitHub at https://github.com/jolly-chen/msc-project/blob/main/microbenchmarks/calibration_microbenchmarks.cu

Note that these microbenchmarks include a warmup call to one of the GPU kernels. This ensures that the initialisation of the CUDA runtime is not included in the benchmark results. In addition, for benchmarks that output a value, we save that value in the state counters. Not only does this allow for error checking, it also prevents the compiler from eliminating the code because it is marked as unused.

Listing C.1: Microbenchmark for calibrating $T_{FindBinGPU}$

```
static void BM_BinarySearchGPU(benchmark::State &state) {
    constexpr long long repetitions = 1000;
    long nbins = state.range(0) / sizeof(double); // Increasing
        histogram size
    size_t bulkSize = state.range(1);
    int blockSize = 256;
    int numBlocks = bulkSize % blockSize == 0 ? bulkSize / blockSize :
        bulkSize / blockSize + 1;

    std::random_device rd; // a seed source for the random number engine
    std::mt19937 gen(rd()); // mersenne_twister_engine seeded with rd()
    std::uniform_int_distribution<> distrib(0, nbins);
    AlignedVector<double, 64> vals(bulkSize);
    for (auto i = 0; i < bulkSize; i++) vals[i] = distrib(gen);
    double *d_vals;
    ERRCHECK(cudaMalloc((void **)&d_vals, bulkSize * sizeof(double)));
    ERRCHECK(cudaMemcpy(d_vals, vals.data(), bulkSize * sizeof(double),
        cudaMemcpyHostToDevice));

    AlignedVector<double, 64> binedges(nbins);
```

```

for (auto i = 0; i < nbins; i++) binedges[i] = i;
double *d_binedges;
ERRCHECK(cudaMalloc((void **)&d_binedges, nbins * sizeof(double)));
ERRCHECK(cudaMemcpy(d_binedges, binedges.data(), nbins *
    sizeof(double), cudaMemcpyHostToDevice));

//warmup
BinarySearchGPU<<<numBlocks, blockSize>>>(nbins, d_binedges, d_vals,
    static_cast<double*>(0));

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
for (auto _ : state) {
    cudaEventRecord(start);
    for (auto i = 0; i < repetitions; i++) {
        BinarySearchGPU<<<numBlocks, blockSize>>>(nbins, d_binedges,
            d_vals, static_cast<double*>(0));
    }
    cudaEventRecord(stop);

    cudaEventSynchronize(stop);
    float elapsed_milliseconds;
    cudaEventElapsedTime(&elapsed_milliseconds, start, stop);
    state.SetIterationTime(elapsed_milliseconds/1e3); // Iteration
        time needs to be set in seconds
}

state.counters["repetitions"] = repetitions;
state.counters["nbins"] = nbins;
state.counters["bulksizes"] = bulkSize;
state.counters["numblocks"] = numBlocks;
ERRCHECK(cudaFree(d_binedges));
ERRCHECK(cudaFree(d_vals));
ERRCHECK(cudaEventDestroy(start));
ERRCHECK(cudaEventDestroy(stop));
}

BENCHMARK(BM_BinarySearchGPU)
    ->ArgsProduct({benchmark::CreateRange(8, 268435456, 2), // Array size
        benchmark::CreateRange(32, 262144, 4), // Bulksizes
    })
    ->Unit(benchmark::kMicrosecond)->UseManualTime()
    ->MinTime(1e-3); // repeat until at least a millisecond since the
        resolution of cudaEventRecord is 0.5 us

```

C. GPU CALIBRATION MICROBENCHMARKS

Listing C.2: Microbenchmark for calibrating $T_{AddBinContentGPU}$

```
static void BM_HistogramGPU(benchmark::State &state) {
    constexpr long long repetitions = 1000;
    long nbins = state.range(0) / sizeof(double); // Increasing
        histogram size
    size_t bulkSize = state.range(1);
    bool global = state.range(2) == 1 ? true : false;
    int blockSize = 256;
    int numBlocks = bulkSize % blockSize == 0 ? bulkSize / blockSize :
        bulkSize / blockSize + 1;
    auto smemSize = nbins * sizeof(double);

    int maxSmemSize;
    cudaDeviceGetAttribute(&maxSmemSize,
        cudaDevAttrMaxSharedMemoryPerBlock, 0);

    double *d_histogram;
    ERRCHECK(cudaMalloc((void **)&d_histogram, nbins * sizeof(double)));

    AlignedVector<int, 64> coords(bulkSize);
    for (auto i = 0; i < bulkSize; i++) coords[i] = 0;

    int *d_coords;
    ERRCHECK(cudaMalloc((void **)&d_coords, bulkSize * sizeof(int)));
    ERRCHECK(cudaMemcpy(d_coords, coords.data(), bulkSize * sizeof(int),
        cudaMemcpyHostToDevice));

    AlignedVector<double, 64> weights(bulkSize, 1);
    double *d_weights;
    ERRCHECK(cudaMalloc((void **)&d_weights, bulkSize * sizeof(double)));
    ERRCHECK(cudaMemcpy(d_weights, weights.data(), bulkSize *
        sizeof(double), cudaMemcpyHostToDevice));

    // Warmup to load the kernel
    if (global)
        HistogramGlobal<<<numBlocks, blockSize>>>(d_histogram, d_coords,
            d_weights, 0);
    else
        HistogramLocal<<<numBlocks, blockSize>>>(d_histogram, 0, d_coords,
            d_weights, 0);
    ERRCHECK(cudaPeekAtLastError());

    float elapsed_milliseconds;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
```

```

if (global) {
    for (auto _ : state) {
        cudaEventRecord(start);
        for (auto i = 0; i < repetitions; i++)
            HistogramGlobal<<<numBlocks, blockSize>>>(d_histogram,
                d_coords, d_weights, bulkSize);
        cudaEventRecord(stop);
        ERRCHECK(cudaPeekAtLastError());

        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&elapsed_milliseconds, start, stop);
        state.SetIterationTime(elapsed_milliseconds/1e3); // Iteration
                    time needs to be set in seconds
    }
} else {
    if (smemSize < maxSmemSize) {
        for (auto _ : state) {
            cudaEventRecord(start);
            for (auto i = 0; i < repetitions; i++)
                HistogramLocal<<<numBlocks, blockSize,
                    smemSize>>>(d_histogram, nbins, d_coords, d_weights,
                    bulkSize);
            cudaEventRecord(stop);
            ERRCHECK(cudaPeekAtLastError());

            cudaEventSynchronize(stop);
            cudaEventElapsedTime(&elapsed_milliseconds, start, stop);
            state.SetIterationTime(elapsed_milliseconds/1e3); //
                        Iteration time needs to be set in seconds
        }
    } else {
        state.SkipWithError("Does not fit in shared memory");
    }
}

state.counters["repetitions"] = repetitions;
state.counters["nbins"] = nbins;
state.counters["bulksize"] = bulkSize;
state.counters["numblocks"] = numBlocks;
state.counters["global"] = global ? 1 : 0;
ERRCHECK(cudaFree(d_histogram));
ERRCHECK(cudaFree(d_coords));
ERRCHECK(cudaFree(d_weights));
ERRCHECK(cudaEventDestroy(start));
ERRCHECK(cudaEventDestroy(stop));
}
BENCHMARK(BM_HistogramGPU)

```

C. GPU CALIBRATION MICROBENCHMARKS

```
->ArgsProduct({benchmark::CreateRange(8, 268435456, 2), // Array size
              benchmark::CreateRange(32, 262144, 4), // Bulksize
              {1, 0}, // global, local
            })
->Unit(benchmark::kMicrosecond)
->UseManualTime()
->MinTime(1e-3); // repeat until at least a millisecond since the
                resolution of cudaEventRecord is 0.5 us
```

Listing C.3: Microbenchmark for calibrating $T_{UpdateStatsGPU}$

```
static void BM_TransformReduceGPU(benchmark::State &state) {
    constexpr long long repetitions = 10000;
    size_t numElements = state.range(0);
    int blockSize = state.range(1);
    int numThreads = (numElements < blockSize * 2) ?
        nextPow2((numElements + 1) / 2) : blockSize;
    int numBlocks = (numElements + (numThreads * 2 - 1)) / (numThreads *
        2);

    AlignedVector<double, 64> data(numElements);
    for (auto i = 0; i < numElements; i++) data[i] = i;
    double *d_data;
    ERRCHECK(cudaMalloc((void **)&d_data, numElements * sizeof(double)));
    ERRCHECK(cudaMemcpy(d_data, data.data(), numElements *
        sizeof(double), cudaMemcpyHostToDevice));

    double *d_out;
    ERRCHECK(cudaMalloc((void **)&d_out, sizeof(double)));

    // warmup
    TransformReduce(numBlocks, blockSize, numElements, d_out, 0., true,
        Plus{}, Identity{}, d_data);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    for (auto _ : state) {
        cudaEventRecord(start);
        for (auto i = 0; i < repetitions; i++) {
            TransformReduce(numBlocks, blockSize, numElements, d_out, 0.,
                true, Plus{}, Identity{}, d_data);
        }
        cudaEventRecord(stop);

        cudaEventSynchronize(stop);
        float elapsed_milliseconds;
        cudaEventElapsedTime(&elapsed_milliseconds, start, stop);
    }
}
```

```

    state.SetIterationTime(elapsed_milliseconds/1e3);
}

state.counters["repetitions"] = repetitions;
state.counters["numelements"] = numElements;
state.counters["numblocks"] = numBlocks;
state.counters["numthreads"] = numThreads;
state.counters["blocksize"] = blockSize;
ERRCHECK(cudaFree(d_data));
ERRCHECK(cudaFree(d_out));
ERRCHECK(cudaEventDestroy(start));
ERRCHECK(cudaEventDestroy(stop));
}

BENCHMARK(BM_TransformReduceGPU)
->ArgsProduct({
    benchmark::CreateRange(32, 262144, 8), // array size
    {256}, // blockSize
})
->Unit(benchmark::kMicrosecond)
->UseManualTime()
->MinTime(1e-3); // repeat until at least a millisecond since the
                 resolution of cudaEventRecord is 0.5 us

```

Listing C.4: Microbenchmark for calibrating $T_{MemCpy_{HTOD}}$

```

static void BM_MemcpyCPUToGPU(benchmark::State &state)
{
    constexpr long long repetitions = 300;
    auto nbytes = state.range(0);
    bool pinned = state.range(1) == 1 ? true : false;

    void *data;
    if (pinned)
        ERRCHECK(cudaMallocHost((void **)&data, nbytes));
    else
        data = malloc(nbytes);

    void *ptr;
    ERRCHECK(cudaMalloc((void **)&ptr, nbytes));

    // Warmup
    cudaMemcpy(ptr, data, nbytes, cudaMemcpyHostToDevice);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    for (auto _ : state) {
        cudaEventRecord(start);

```

C. GPU CALIBRATION MICROBENCHMARKS

```
// auto start = Clock::now();
for (auto i = 0; i < repetitions; i++)
    cudaMemcpy(ptr, data, nbytes, cudaMemcpyHostToDevice);
cudaEventRecord(stop);
// auto end = Clock::now();

cudaEventSynchronize(stop);
float elapsed_milliseconds;
cudaEventElapsedTime(&elapsed_milliseconds, start, stop);
state.SetIterationTime(elapsed_milliseconds/1e3);

// auto elapsed_seconds = std::chrono::duration_cast<fsecs>(end -
// start);
// state.SetIterationTime(elapsed_seconds.count());
}

state.counters["nbytes"] = nbytes;
state.counters["pinned"] = pinned ? 1 : 0;
state.counters["repetitions"] = repetitions;
cudaFreeHost(data);
cudaFree(ptr);
ERRCHECK(cudaEventDestroy(start));
ERRCHECK(cudaEventDestroy(stop));
}
BENCHMARK(BM_MemcpyCPUToGPU)
->ArgsProduct({benchmark::CreateRange(1, 33554432, 2), // Array size
              {1, 0}, // pinned, pageable
})
->ArgsProduct({benchmark::CreateDenseRange(33554432, 268435456,
      int(268435456-33554432)/10), // Array size
              {1, 0}, // pinned, pageable
})
->MinTime(1e-3) // repeat until at least a millisecond since the
               // resolution of cudaEventRecord is 0.5 us
->UseManualTime()->Unit(benchmark::kMicrosecond);
```

Acknowledgements

Completing this thesis has been a lengthy journey, and it would not have been possible without the support of many.

First and foremost, I extend my deepest gratitude to my supervisors, Monica Dessole and Ana Lucia Varbansecu. I cannot thank them enough for their invaluable guidance and the time they dedicated to reviewing this work.

At CERN, being part of the ROOT team has been an unforgettable experience. Over the past year, I have had the privilege to work with some one of the most knowledgeable, inspirational, and kind-hearted individuals I have met. I want to especially thank Axel Naumann for giving me this opportunity and for his supervision at the beginning.

Lastly, I am extremely grateful to my parents, Bridget, Emma, and Shenglin for their unwavering support and encouragement in all my adventures. I would also like to acknowledge Rhys for fuelling my Thursdays with his (almost) weekly brioche deliveries.