Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Using GPUs as Decompression Accelerators in ROOT Applications

Author: Ruben Stap (rsp480)

1st supervisor: prof.dr.ir. A.L. Varbanescu
2nd reader: dr. M. Dessole

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

April 7, 2024

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Context. at the end

Goal. at the end

Method. at the end

Results. at the end

Conclusions. at the end

Contents

1

Introduction

In today's day and age, widespread ownership of powerful computational devices across users and organisations, has allowed the inception of many new commercial services, such as information retrieval systems, e-commerce, streaming services, recommendation systems and fraud detection systems. These services rely on huge volumes of data, for their usability and/or profitability. The currently available computational power, has also allowed research to incorporate analyses on huge volumes of data. This has occurred in diverse fields such as tourism, health care, education and high energy physics. We will now exemplify these new commercial services, as well as research analysing huge volumes of data, starting with commercial services.

An example retailer that also participates in e-commerce, is Wallmart. Due to this, Wallmart can offer an estimated 50 to 60 million products to its customers (1). In companies such as Wallmart, sale drops can obviously occur. In order to quickly analyse and resolve those sale drops, Wallmart built a system capable of analysing a data-set including at least 40000 terabytes of recent transactional data. Apart from this, Wallmart also offers a product recommendation system (2), that potentially increases their sales.

Streaming services such as Spotify and Youtube music both offer more than 100 million tracks, and also employ recommendation systems, albeit for music (3, 4, 5). Popular information retrieval systems such as Google, also offer relevant content, in the form of a listing of web pages, on the basis of a search query. The inception of such information retrieval systems enables users to quickly navigate a web of a currently estimated 50 billion to hundreds of billions web pages (6).

All the aforementioned services and many other businesses use systems enabling monetary transactions. Zooming in on credit cards alone, reveals that customers used them in an estimated 678 billion transactions during 2022 (7). Obviously, those transactions can

be fraudulent. As such, all major credit card suppliers employ systems which act upon huge volumes of transactional data, to detect fraudulent transactions (8), and to minimise corresponding losses. Even with the presence of fraud detection systems, "The Nilson Report" estimates that credit card companies will lose 408.5 billion on fraud, between 2020 and 2030 (9).

Moving on to exemplifying the analyses of huge volumes of data in research, remember that we mentioned that such analyses occur in diverse fields. For example, in the medical field, researchers discovered a potential treatment, for an aggressive form of brain stem cancer, by processing over 40 million documents (10, 11). In the entirely different field of tourism, researchers used a dataset of millions of foreign cell phone usage logs, to develop a process for the analysis of visitor flows to various destinations (12). This process paves the way for companies to improve the packaging of travel destinations, in addition to improving their targeting towards interested customers. Millions of dataset entries, originating from an educational system consisting of a classical learning management system, an e-portfolio system, and a system offering e-books, also drove a quantitative study on the effect of student behavior on academic performance (13).

At last we will discuss the usage of huge volumes of data in the research field of *High Energy Physics* (HEP). In HEP, researchers study the elementary constituents of matter using high energies, hence its name (14). CERN performs several big experiments contributing to HEP (15). For this it uses the *Large Hadron Collider* (LHC) located near Geneva, which is a 27 km long ring, where particles accelerate to high speeds and thus gain high energy, until they collide at four particle detectors named as ATLAS, ALICE, CMS and LHCB (16, 17). These detector names are also the names of the corresponding experiments, and do not cover all of the 9 experiments, currently performed using the LHC (18). The four mentioned experiments alone, produce up till 88000 TB of data each year, an incredible volume of data to process, store, and analyse (19).

Up till now we have not yet discussed the current computational architectures which enable research and commercial services, to utilise huge volumes of data. Nowadays, such architectures consist of one or more interconnected nodes, which can consist of one or more of the following components: *central processing units* (CPUs), *graphical processing units* (GPUs), *field programmable gate arrays* (FPGAs), and *application specific integrated circuits* (ASICs) (20). Note that this list is not meant to be exhaustive. Whereas current multi-core CPUs allow efficient parallel execution of complex instruction streams across cores, GPUs consist of many more cores, thus enabling higher computational throughput, on the condition of a simple instruction stream per core.

The other mentioned chips tailor the hardware specifically to an application, whereby FPGAs essentially allow one to program a custom chip by enabling and interconnecting components available on the FPGA, whereas ASICs require custom design and fabrication, a process that is extremely expensive. All available components in such nodes, point to the current heterogeneity of computing nodes. The trend of chip specialisation, as well as increasing core counts in CPUs, is expected to continue in computing architectures, because it is the only remaining axis of potential performance improvements. Optimal usage of the available computing components in applications, is crucial for them to achieve good performance.

Nowadays, applications are oftentimes bound by memory bandwidth, which is a result of the current rift between computational and memory bandwidth, which we only expect to increase in the near future. Therefore, minimising memory requirements in such applications is a factor crucial for them to attain good performance. Usage of compression on the datasets at hand can serve the end of minimising data requirements. Performance is then improved because the process of potentially compressing, transferring and decompressing data has a higher bandwidth than transferring uncompressed data alone. Obviously, data compression also allows a more efficient usage of storage space, reducing its cost especially at huge dataset sizes. Exactly because of both reasons, CERN uses data compression across its previously described LHC experiments (21).

In a scenario where we use huge datasets in an application, any mapping of application components to computing components, can potentially reach top performance. Obviously this depends on the datasets at hand, whether we need to decompress data and if so the used compression algorithm. Zooming in on data decompression, CPUs and GPUs, there is evidence that GPUs can outperform CPUs for decompression (22, 23, 24). This advantage of GPU decompression demonstrates the possible usefulness of introducing GPU decompression in CPU-only applications. Additionally, using GPU decompression in GPU-only applications (25, 26, 27, 28) or hybrid GPU-CPU applications (29) can also lead to better performance. Any application processing data on the CPU, GPU or both, can thus potentially benefit from using the GPU for decompression. In this thesis we will focus on predicting the effectiveness of optimising applications which use compressed data, that currently only use a single CPU, by offloading decompression to the GPU. More specifically, we will describe a process with which we can make such an prediction. In other words, we will provide an answer to the following research question.

RQ: How can we optimise decompression in an application with the GPU?

One can apply the resulting process, as well as other reusable components provided by this thesis, to any CPU application satisfying the mentioned requirements, in order to evaluate the effectiveness of GPU decompression offloading, without requiring an implementation.

By answering this research question for a specific set of applications, we could distill a process answering it in general. We looked at applications written using the ROOT application framework, that is used in analysing the huge amounts of (compressed) experimental data produced by almost all LHC experiments from CERN. This framework consists of built-in graphing routines, common analysis operations such as the Fourier transform, and a data persistence mechanism (30).

One can write ROOT applications in different idioms. To bound project complexity, we chose to focus on the most popular RDataFrame idiom. Conceptually, these programs operate on certain columns of a tabular dataset. Fundamental operations in these programs constitute the selection of table rows, the definition of new column values, and the aggregation of column values such as finding a average. Such a program can be as simple as `dataset.Filter("b > 2").Define("d = c * c").Sum("d")` where we first select all table rows where column b has a value greater than 2, then define a new column d as the square of c, and finally sum the newly created column which serves as program output.

The process answering the research question involves the creation of models. Within this thesis, we define a model as a function $f(x_1, \ldots, x_n)$ that can make predictions. Such a model can make predictions, as a function of variables $x_1, x_2 \ldots, x_n$.

1.1 Thesis Outline

As a first step in answering the research question, we created a latency model for RDataFrame applications in general, to gain enough understanding for initiating the evaluation of GPU decompression approaches. Before describing this step, the upcoming chapter will first of all describe some required background information.

2

Background

2.1 Analytical models

As discussed in the introduction, we define an analytical model, as a mathematical expression that relates N input variables $x_1, x_2, \ldots, x_N$ with an output variable y. In general terms, one can only refer to such an expression as a functional model, when it characterises a system under some conditions. In this thesis, we bound these systems to application components, and we characterise these components by predicting metrics such as a latency or throughput.

We consistently follow a certain methodology when creating and readying such a model to make predictions, which we will now discuss using an example. This example model will predict the latency of an imaginary application component. Most analytical models in this thesis will predict latency values.

2.1.1 Creating the expression

As a first step, we must create the mathematical expression that characterises the latency of an application component. In actuality, one would base this expression on an understanding of the application component. In the absence of a real application component, we will assume that the following linear expression $f(n) = a \cdot n$ results from such a deep understanding.

2.1.2 Calibration

Before this model can make predictions, one should set the value of a, with a process we refer to as calibration, training or fitting. Intuitively, we refer to a model resulting from this process as a calibrated, trained or fitted model. In general, this process sets all

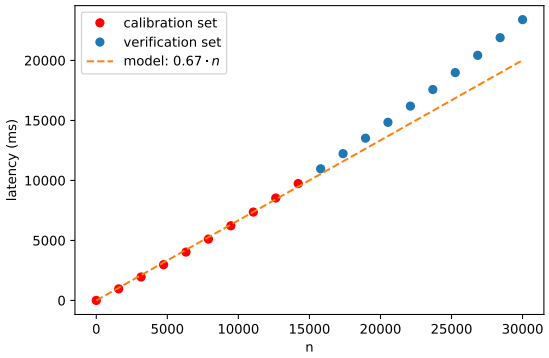


Figure 2.1: All points represent measured latencies of the application component, for certain input sizes. Using the least squares method, we fitted $f(n)$ to all red points in the so-called calibration set. The graph displays this fitted model with the dotted line.

model variables that are no input variables, using a set of n-tuples, which we refer to as a calibration or training set.

Each red point in figure 2.1 represents the measured latency for an input size. All these points together constitute a calibration set for the imaginary application component. We used the least squares method on this set, to create a calibrated model which we visualise with the orange dotted line. This method sets a to minimise the sum of the squared distances between each measured latency in the calibration set and the predicted latency for the same n (31).

2.1.3 Verification

After calibrating the model, we need to check how well it can predict the latency of the application component. When each n-tuple in the calibration set contains input variable values and corresponding output variable measurements, one can start the accuracy evaluation on this set. Given that this is the case for our example model, we proceed.

Visually, figure 2.1 indicates that the model predictions are close to the measured calibration set values. Numerically, the calculated quotients M/R between model predictions M and real latency measurements R in the calibration set, indicate that the model overestimates and underestimates the real measurements by at most 9.5% and 2.6%, respectively. These deviations lie just within the reasonable bound of 10% that we use. If the deviations did not lie within this bound, we would return to the first step of this process, which encompasses studying the application component at hand, to improve the model.

It is also commonplace, to evaluate a models accuracy on a set of tuples that consists of input variable values and corresponding output variable measurements, that do not overlap

with the calibration set. We call such a set, a verification or a test set. Doing this, we can get some info on the generalisability of the model beyond input variable values in the calibration set.

More formally, the meaning of a verification set not overlapping with a calibration set, is as follows. There may not be tuples in the verification set for which there are tuples in the calibration set, with the same input variable values. All blue points in figure 2.1 form a verification set, because their n-values do not overlap with points in the calibration set.

Visual inspection of the distance between points in the verification set and the model predictions reveals that the model underestimates the latency much more than in the calibration set. Given that we determined the maximum overestimation for the calibration set to be 9.5%, this underestimation is much bigger than our reasonable bound. When we determine that a measurement error did not cause this underestimation, we should return to the first step of this process, to create a more accurate model.

2.2 Related work

This work is the first to address fine-grained, detailed modeling of ... This has been inspired by the work of Gorgan ... However, their models are much less detailed.

- Compression and decompression algorithms and their performance on CPU and GPU
- Modeling for selecting compression/algorithms
- Models for selecting CPU or GPUs
- I/O structure and performance

Work in progress

3

First models

In this chapter, we will build increasingly more fine grained models that capture the current execution latency of RDataFrame programs. We will calibrate these models for the LHCB benchmark, a program provided by CERN (32). This program creates a mass histogram for the so-called B-meson particle.

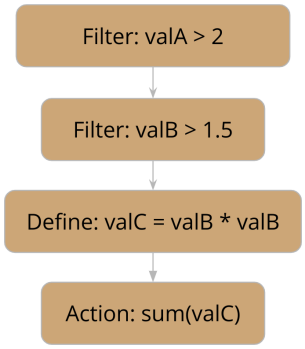
3.1 RDataFrame datasets

An RDataFrame program processes a dataset. For now, we conceptualize such a dataset as tabular. We refer to rows of this table as events, whereas we refer to table columns as fields. The rightmost table in Figure 3.1 displays a simple example dataset of 1000 events with fields valA and valB, respectively. Underneath this abstraction, there are various dataset storage schemes, such as TTree (33) or its most recent extension RNTuple (34). Within this thesis, we will use the RNTuple scheme, as this makes the insights relevant and future-proof.

3.2 RDataFrame programs

RDataFrame programs consist of one or more chains, of actions and/or transformations (35). We will describe the exact manner in which these chains work, as well as elaborate on these operation classes, by looking at a simple example chain displayed in Figure 3.1. This chain consists of 2 filters, 1 define and an action. Mentally, imagine that all events of the dataset flow through this chain, starting at the first filter.

The first filter, selects all events where valA is bigger then two. These events then enter the second filter, which selects all events where valB is bigger than 1.5. Events where valA



event no #	valA	valB
0	9	3.0
1	6	2.5
2	13	1.3
...	...	...
1000	12	1.5

Figure 3.1: The flowchart visualises an example RDataFrame program, whereas the table displays the dataset on which it acts. The program makes a selection of events, subsequently calculates the square of valB of these events, and then computes the sum of these squared values.

is bigger than two and valB is bigger than 1.5, pass both filters. For each of those events, we then define a new field valC as the square of valB. The final action sums all values in this newly defined column.

Filters and defines belong to the transformation class of RDataFrame operations. To bound project complexity, we will only develop models that capture those transformation operations. Apart from the summation action, one can also perform other aggregating actions such as computing a mean, standard deviation, minima, or something user-defined. One can also create graphs, and histograms, that visualise (newly computed) dataset fields.

3.2.1 One or more event loops

The RDataFrame framework determines the result of an operation chain by looping over all events in the dataset. We refer to this as an event loop. For the simple operation chain shown in Figure 3.1, the framework determines the summed field valC by evaluating the chain, for each event separately. For an event, this encompasses checking whether the event passes the first and second filter, if so computing valC, and finally adding it to a sum variable. When an event does not pass either filters, the event loop simply starts evaluating the next dataset event.

As mentioned earlier, RDataFrame programs can consist of multiple operation chains. If possible, the RDataFrame framework executes these chains within a single event loop. This minimises the required amount of field value fetching with respect to computation, making it an ideal situation in terms of performance. The following example RDataFrame program, produces two histograms, for a sampled random variable $X \sim N(0, 1)$ and the random variable $Z = X \cdot 2$, respectively.

```
auto df = ROOT::RDF::FromCSV("normal_rv.csv");
auto c1 = new TCanvas("x", "X ~ N(0,1)", 800, 600);
auto histo = df.Histo1D({"x_rv", "X ~ N(0,1)", 64u, -5.0, 5.0}, "x");
histo->DrawCopy("", ""); // Draw histogram for X

auto c2 = new TCanvas("z", "2 * X with X ~ N(0,1)", 800, 600);
auto df1 = df.Define("z", "x*2");
auto histo1 = df1.Histo1D({"z_rv", "2 * X with X ~ N(0,1)", 64u, -10.0, 10.0}, "z");
histo1->DrawCopy("", ""); // Draw histogram for Z
```

In this code, we first load the dataset in `df`. In this example, we read from a csv dataset, consisting of one column of samples of $X \sim N(0, 1)$. With the `Histo1D` action, we book the generation of a histogram of this column, with 64 bins from -5 to 5. Actions do not generate results immediately, but generate a pointer to their results. Whenever we ask for the value of such a result, or in this case, an actual graph with the `DrawCopy` call, the RDataFrame framework tries to evaluate all execution chains up till that point, with a single event loop. Because the program declares the define and second histogram generating operation after the first `DrawCopy` call, a second event loop performs those operations. All in all, this program needs two event loops for its execution.

```
auto df = ROOT::RDF::FromCSV("normal_rv.csv");
auto histo = df.Histo1D({"x_rv", "X ~ N(0,1)", 64u, -5.0, 5.0}, "x");
auto df1 = df.Define("z", "x*2");
auto histo1 = df1.Histo1D({"z_rv", "2 * X with X ~ N(0,1)", 64u, -10.0, 10.0}, "z");

auto c1 = new TCanvas("x", "X ~ N(0,1)", 800, 600);
histo->DrawCopy("", ""); // Draw histogram for X
auto c2 = new TCanvas("z", "2 * X with X ~ N(0,1)", 800, 600);
histo1->DrawCopy("", ""); // Draw histogram for Z
```

By reordering the statements so that we only ask to display the graphs after declaring all execution chains, we allow the RDataFrame framework to execute all these chains, within a single event loop.

All developed models in this thesis, will only capture RDataFrame programs that execute in a single event-loop. This assumption correponds with the intended use-case of RDataFrame programs. We made this assumption to bound project complexity, for which we will provide more rationale, in the next chapter.

3.3 LHCB benchmark

Introductory remarks in this chapter already mentioned the LHCB benchmark, which produces the mass histogram of a certain particle referred to as the B-meson. This program

operates on the B2HHH dataset of 8.5 million events, and 26 simple double and integer fields (36). Note that the LHCB benchmark can operate on this dataset in both the TTree and RNTuple format. One can convert the dataset from the TTree format to the RNTuple format, using `gen_lhcb` from the iotools repository (37).

The LHCB benchmark contains code for computing the mass histogram in two different programming idioms. Apart from containing code for computing the mass histogram in the RDataFrame idiom, it also contains code for computing it in another idiom. In the latter idiom, one programs all event loops manually.

Figure 3.2 shows the event-loop underlying the execution of either idioms. It is alike the simple RDataFrame program explained before, being more complex, however. More specifically, if one inspects this diagram in more detail, it reveals 9 filters, 9 defines and 1 histogram adding operation. All brown arrows indicate data dependencies, coming from a previous define or the B2HHH dataset itself.

3.4 Model - single execution path terms

3.4.1 Description

Remember that we only model RDataFrame programs which need a single event loop for their execution. Such programs consist of m filters, and an arbitrary number of actions and defines. Figure 3.3 visualises all operations in the event loop of such a program. Each event loop iteration has $m+1$ possible execution paths. When we refer to execution path P_i with $1 \leq i \leq m$ we refer to the execution path where execution does not pass filter i. Obviously, execution then passed all previous filters and all other preceding operations. In the final path P_{m+1}, execution passes all filters and also executes any possible leftover operation. In Figure 3.3 this refers to the white boxes succeeding filter m. When assuming that each of these execution paths P_i has a fixed execution latency T^i_{path} , and by additionally assuming that the fraction f^i_{path} of all events N_{events} which execute in this path, remains constant across different dataset sizes, we can model the event loop latency with the following equation.

$$T_{\text{eventloop}}(N_{\text{events}}) = \sum_{i=1}^{m+1} N_{\text{events}} f^i_{\text{path}} T^i_{\text{path}}$$

Obviously, the field value distributions influence the parameters f^i_{path}. When calibrating this model, one can thus only expect it to be accurate, for datasets with equivalent field value distributions. We hypothesize that all datasets that are meant to be used with an RDataFrame program, satisfy this criterion. For convenience, we refer to such datasets, as

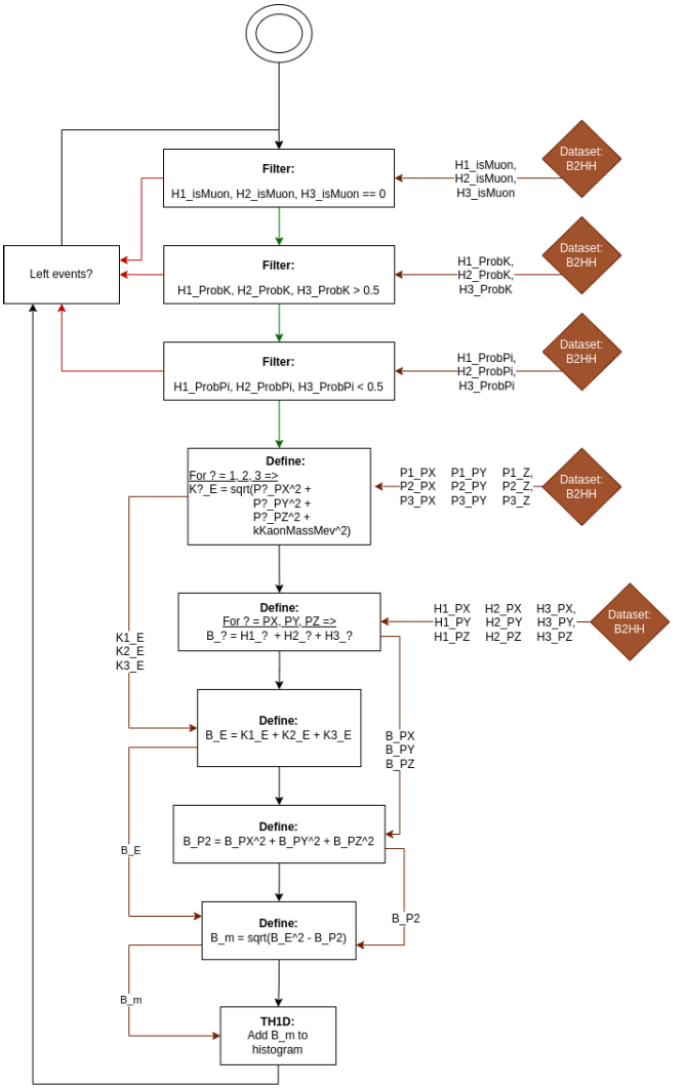


Figure 3.2: Displays the execution of both the RDataFrame and the manual event-loop idiom, at the level of the underlying event-loop. All brown arrows indicate data dependencies.

datasets of a certain type. Datasets of a certain type thus consist of equivalent field value distributions, apart from containing the same fields in terms of their name and type.

3.4.2 Calibration

In order to calibrate this model, we measured the distribution of event loop iteration latencies, for the LHCB benchmark, given the B2HHH dataset, without any compression. We created this RNTuple dataset from the provided B2HHH dataset with zstd compression, using the `gen_lhcb` tool from the iotools repository.

Given that the LHCB benchmark consists of 9 filters, it has 10 different execution paths.

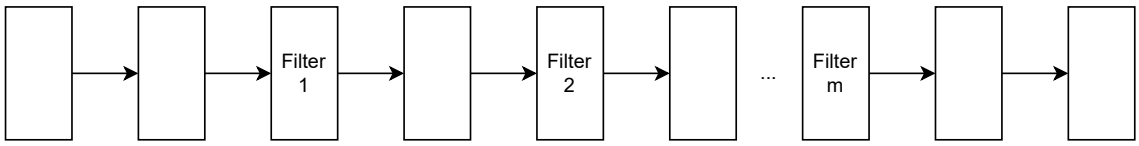


Figure 3.3: Visualises all potentially executed operations in an event loop iteration of a single event loop RDataFrame program. Each empty block corresponds to an action or a define.

We thus expected to find 10 distinct peaks in the distribution of event loop iteration latencies. The latency value and frequency of a peak, would then define a f^i_{path} and T^i_{path} for an execution path i.

3.4.2.1 Coupling the event loop iteration latency with code

The RDataFrame framework, executes event loops in one of the functions in `RLoopManager.cxx`, depending on the amount of used threads, the chosen interface and dataset storage scheme, as well as other criterion (38). Given that we currently study the RDataFrame framework in single-threaded mode, for a RNTuple datset, this is the `RLoopManager::RunDataSource` function.

```
fDataSource->Initialize();
auto ranges = fDataSource->GetEntryRanges();

while (!ranges.empty() && fNStopsReceived < fNChildren) {
   InitNodeSlots(nullptr, 0u);
   fDataSource->InitSlot(0u, 0ull);
   RCallCleanUpTask cleanup(*this);
   try {
      for (const auto &range : ranges) {
         const auto start = range.first;
         const auto end = range.second;
         for (auto entry = start; entry < end && fNStopsReceived < fNChildren; ++entry) {
            if (fDataSource->SetEntry(0u, entry))
               RunAndCheckFilters(0u, entry);
         }
      }
   } catch (...) {
      std::cerr << "RDataFrame::Run: event loop was interrupted\n";
      throw;
   }
   fDataSource->FinalizeSlot(0u);
   ranges = fDataSource->GetEntryRanges();
}

fDataSource->Finalize();
```

The preceding code, corresponds to this function. Initially, it retrieves the event range over which it must iterate. For our RNTuple datasource, this is just a single tuple, which `GetEntryRanges` only returns once(39). As such, the framework only spends one iteration in the while loop and its immediate descending for loop. The framework thus actually loops over all events, in the innermost for loop. Therefore we assumed that the framework performs most of the work here, and we took the latency distribution of this for loop as the latency distribution of the entire event-loop. Apart from rationale given before, the fact that `InitSlot`, `FinalizeSlot` and `Finalize` are void for an RNTuple datasource, also gives evidence for this assumption (40, 41, 42, 43, 44). In actuality, we didn't measure the latency distribution of the entire loop body, but rather that of `RunAndCheckFilters`, as we assumed that this is the most dominant loop iteration latency.

3.4.2.2 Measuring the event loop iteration latency

We measured the `RunAndCheckFilters` latency distribution, by surrounding this function with `__rdtsc()` calls, that return the CPU clock cycle count. Initially, we also tried other timers such as TStopWatch provided by ROOT (45), and std::chrono::system_clock (46), but these turned out to significantly increase execution time. We saved the cycle difference, between both CPU clock cycle counts, in an unordered map (47), which is a C++ standard library hashmap. Storing each iteration cycle count in a fixed-size array or vector resulted in a higher instrumentation overhead. At the end of the `RunDataSource` function, we finally print the contents of this hashmap on `stderr`.

We performed all initial experiments on Amazon EC2 instances and a local device. All devices ran at a fixed clock frequency to enable the conversion of CPU clock cycles to time. It is important to note, that before we could read the `RunAndCheckFilters` latency distribution on `stderr`, after running the `lhcb` benchmark, we needed to recompile the ROOT framework with the added instrumentation. Obviously, another prerequisite was the recompilation of the `lhcb` program with this modified framework.

3.4.2.3 Experiment descriptions

We used an c4.8xlarge Amazon EC2 instance underlayed by an Intel E5-2660 v3 processor running at 2.9 GHZ, for all final measurements in this chapter. More specifically, we determined the final latency distribution of `RunAndCheckFilters`, in the LHCB benchmark applied on the RNTuple B2HHH dataset without compression. Apart from purely measuring the `RunAndCheckFilters` latency distribution, we also measured the latency of the

entire `RunDataSource` function, again using `__rdtsc()` calls. For both experiments, and all others in the context of this first model, we repeated each measurement 10 times, and we pinned all execution threads to a separate core using `likwid-pin` unless designated otherwise. We did the latter to prevent variability and a potential latency penalty due to thread migration. We also cleared operating system caches between each measurement, using `echo 3 > /proc/sys/vm/drop_caches`, to prevent an unfair advantage of some measurement repetitions, due to a (partially) cached dataset. In this way, each measurement accurately captures running the benchmark on the B2HHH dataset for the first time.

The LHCB benchmark measures a so-called analysis time, and initialization time, by default. The analysis time is the time between the start of the application of the first filter on the first event, and the full materialization of the requested histogram. On the other hand, the initialization time is the time between the creation of the datasource object that encapsulates the B2HHH dataset, and the starting point of the analysis time. We also determined these time values using the ROOT framework without instrumentation, to say something about the instrumentation overhead.

3.4.2.4 Results & discussion of the experiments

Without instrumentation, the sum of the analysis and initialization time is 10.51 seconds on average (var=0.0410). With instrumentation, this sum is 10.55 seconds on average (var=$1.483 \cdot 10^{-3}$). The instrumentation itself thus seems to have a negligible latency. Additionally, the RunDataSource function itself, disregarding the time to print and create the unordered map, takes 10.51 seconds (var=$1.152 \cdot 10^{-3}$). The RunDataSource function thus seems to accurately capture the sum of the analysis and initialization time.

Remember that we drop operating system caches before taking any measurement. When not doing so, we get a mean sum of 1.05 seconds (n=100, var=$2.27 \cdot 10^{-2}$) for the analysis and initialization time taken together. This dramatic increase demonstrates the necessity of this measure, to accurately capture the latency of running the benchmark the first time.

When we sum all individually measured loop iteration latencies, we obtain an average of 10.29 seconds (var=$2.020 \cdot 10^{-2}$). This is fairly close to the average analysis time with and without instrumentation, which have values of 10.35 seconds (var=$2.897 \cdot 10^{-2}$) and 10.28 seconds (var=$4.1 \cdot 10^{-2}$), respectively. It makes sense that these times lie closer to the summed loop iteration latency than the initialization and analysis times together, as their starting points are now equal. More specifically, the analysis time measurement starts within the first `RunAndCheckFilters` call, as does the first loop iteration latency, whereas the LHCB program starts measurement of the initialization time, before that point.

Based on visual inspection, we consider the `RunAndCheckFilters` call latency distributions to be constant across repeated measurements. The interested reader can check section 7.1 in the appendix, to confirm this claim using the original distributions.

All these distributions are alike the one shown in Figure 3.4, which shows the distribution corresponding to the first repetition, at three different zoom-levels. These distributions have the highest frequencies, at very low time values. Calling the `Report` action on the final filter result, revealed that only 0.3% of all events, pass all filters. In order for `Report` to provide results, we gave all filters in the LHCB benchmark a name (48), which is simply an extra argument to each Filter call. The low acceptance ratio of events by all filters, causes the many low latency `RunAndCheckFilter` calls in the distribution.

In this low-latency section, we can identify four peaks, of which one can spot three in the middlemost graph. After zooming in, as in the rightmost graph, one can observe the fourth peak. When increasing the latency beyond this peak, the frequencies seem to decrease, and eventually we enter a region of higher but very sparsely occurring `RunAndCheckFilters` latencies.

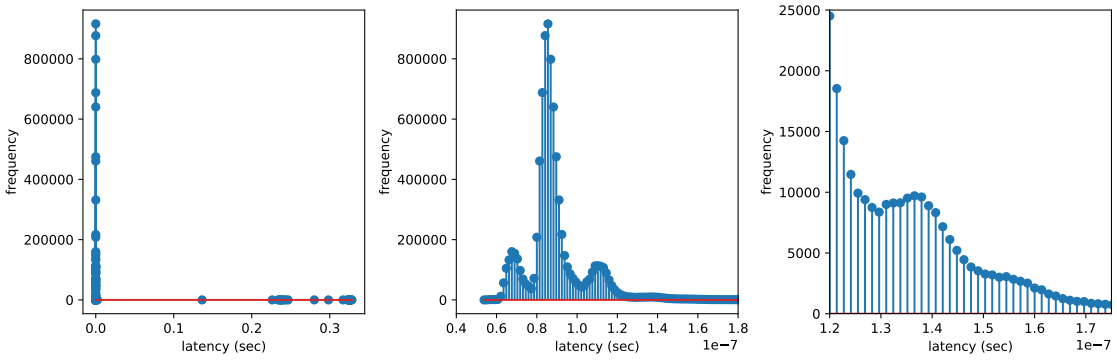


Figure 3.4: Displays the RunAndCheckFilters latency distribution for the first run of the LHCB benchmark, on the B2HHH dataset without compression, using a c4.8xlarge Amazon EC2 instance, for three different latency and frequency ranges.

Contrary to our expectations, we only observe 4 peaks, while we expected 10 peaks. We formulated the following theory as to why this is the case. Previously, we determined that the framework spends the majority of its time in execution paths where a filter eventually rejects an event. Given the simplicity of all filters, they have a very small latency, which lie very close to one another. As a result, the 9 latency peaks, only consisting of filter evaluations, might simply overlap in Figure 3.4, which prevents our observation of them.

3.4.2.5 Next steps

This result indicates a logical next step, in which we determine a latency distribution for each execution path separately, instead of determining a latency distribution for all event loop iterations. Additionally, we will model the time to fetch field values for each execution path, separately. As of now, each execution path latency encompasses both this latency as well as the latency to do the required computation.

3.5 Model - execution path terms split in fetch and computation time

3.5.1 SetEntry

Even though the previous model could capture the analysis time accurately, we suspected a relation between field value fetching and the `SetEntry` call, which was not included in this model. We hypothesized that the exclusion of this call might lead to modelling inaccuracies for datasets bigger than the B2HHH dataset.

This call acts on a so-called RDataSource object, that encapsulates datasets stored using various possible dataset storage schemes such as RNTuple and TTree. Given that we only study datasets stored using the RNTuple scheme, we use a specific RDataSource implementation referred to as RNTupleDS. In this implementation, the `SetEntry` call simply returns true (49). Its triviality as well as the previous model accuracy in predicting the analysis time, allows us to conclude that adding a term to the model capturing its latency, will never improve accuracy by a large margin.

3.5.2 Searching in RunAndCheckFilters for field value fetching

Regardless of this discovery, we still wanted to capture the latency of fetching field values in our model separately, and thus embarked on an investigation on where this occurs specifically within the execution of RDataFrame programs. Given that the `RunAndCheckFilters` latency sum turned out to accurately predict analysis time, we suspected that it occurs within this function.

```
void RLoopManager::RunAndCheckFilters(unsigned int slot, Long64_t entry) {
   if (fNewSampleNotifier.CheckFlag(slot)) {
      for (auto &callback : fSampleCallbacks)
         callback.second(slot, fSampleInfos[slot]);
      fNewSampleNotifier.UnsetFlag(slot);
   }
```

```
   for (auto *actionPtr : fBookedActions)
      actionPtr->Run(slot, entry);
   for (auto *namedFilterPtr : fBookedNamedFilters)
      namedFilterPtr->CheckFilters(slot, entry);
   for (auto &callback : fCallbacks)
      callback(slot);
}
```

In the execution of the LHCB benchmark, the framework only enters the second and third loop, looping over all pointers in `fBookedActions`, and named filters. We determined this by placing print statements in all loops. The fact that the framework enters the third forloop, is intuitive given our modification of the LHCB benchmark, that uses named filters instead of unnamed filters. Remember that this modification enabled the usage of the `Report` function in the calibration of the previous model.

Whereas we previously explained the two operation categories of actions and transformations, consistent with the documentation of RDataFrame, actions in the context of this function have a different meaning. Specifically for the LHCB benchmark they refer to all defines and histogram adding operations. From now on we will refer to actions in the context of this function simply using the variable `actionPtr`, whereas we will keep the meaning of the word action unchanged.

With a scope not tied to the LHCB benchmark, each `actionPtr` can refer to `RAction`, `RJittedAction` or `RVariedAction` class instantiations (50, 51, 52) [1]. An `RJittedAction` requires runtime compilation, whereas `RVariedAction` instantiations allow operation chains to operate on multiple versions of a field that one defines using multiple expressions (53, 54). The LHCB benchmark only contains defines using regularly defined functions, which do not need runtime compilation. If one would for example define these functions as strings, runtime compilation would occur. Additionally, the 1D histogram adding operation `Histo1D` does not need runtime compilation, because the LHCB benchmark specifies the type of the data that it visualises. Given that the LHCB benchmark also does not use variation transformations, all `actionPtr` variables point to instances of the `RAction` class. Moving on to the named filter pointers, these can refer to `RFilter` or `RJittedFilter` instantiations (55, 56, 57). Using similar logic as before, we can conclude that these point to `RFilter` instantiations only, within the LHCB benchmark.

[1] Note that we cite two class references of the RActionBase class, for ROOT 6.30 and ROOT 6.28. We cited both references, because we used a ROOT version directly from its repository, lying between version 6.28 and 6.30.

```
template <typename... ColTypes, std::size_t... S>
void CallExec(unsigned int slot, Long64_t entry, TypeList<ColTypes...>, std::index_sequence<S...>) {
  fHelper.Exec(slot, fValues[slot][S]->template Get<ColTypes>(entry)...);
}

void Run(unsigned int slot, Long64_t entry) final {
  if (fPrevNode.CheckFilters(slot, entry))
     CallExec(slot, entry, ColumnTypes_t{}, TypeInd_t{});
}
```

Listing 3.1: Shows two method which the RunAndCheckFilters function calls on RAction objects for an event. Note that RunAndCheckFilters calls CallExec implicitly through Run.

3.5.2.1 RAction and RFilter

Code listing 3.1, displays the `Run` method of the `RAction` class, which the framework calls on `actionPtr` variables, within a loop in `RunAndCheckFilters`. We present another related function in this snippet, too. As one can see in the `Run` method, the framework first ensures that no filters preceding this `RAction`, rejected the current event. Note that the code refers to an event when mentioning an entry. When the framework enters `CallExec`, it fetches all required field values using the templated `Get` function, whereas it performs the `RAction` itself using the surrounding `fHelper.Exec` function.

In the third loop of `RunAndCheckFilters`, the RDataFrame framework repeatedly calls the `CheckFilters` method on pointers to `RFilter` instantiations. We show this method, as well as an related method, in code listing 3.2. Within the `CheckFilters` method, the framework first checks whether it has already performed the filter for this event. If this is the case, a cache returns the result immediately. When the framework has not performed the filter yet, it checks whether a filter preceding the filter in its execution chain, has already rejected. If this is the case, evaluation of the current filter is not necessary for this event. When the framework has not yet evaluated this filter and does not find any preceding filter rejecting the event, it evaluates the filter by calling `CheckFilterHelper`. This method is alike the `CallExec` method for `RAction` class instantiations, because it retrieves the required field values using a templated `Get` function. The framework performs the filter itself in this function, using the surrounding `fFilter` function.

3.5.3 Second model description

We now hypothesize that we can model the latency of an execution path, by the latency of all filters, defines and actions that it contains, whereby we split each of those latencies in

```
bool CheckFilters(unsigned int slot, Long64_t entry) final {
  if (entry != fLastCheckedEntry[slot * RDFInternal::CacheLineStep<Long64_t>()]) {
     if (!fPrevNode.CheckFilters(slot, entry)) {
        fLastResult[slot * RDFInternal::CacheLineStep<int>()] = false;
     } else {
        auto passed = CheckFilterHelper(slot, entry, ColumnTypes_t{}, TypeInd_t{});
        passed ? ++fAccepted[slot * RDFInternal::CacheLineStep<ULong64_t>()]
               : ++fRejected[slot * RDFInternal::CacheLineStep<ULong64_t>()];
        fLastResult[slot * RDFInternal::CacheLineStep<int>()] = passed;
     }
     fLastCheckedEntry[slot * RDFInternal::CacheLineStep<Long64_t>()] = entry;
  }
  return fLastResult[slot * RDFInternal::CacheLineStep<int>()];
}

template <typename... ColTypes, std::size_t... S>
bool CheckFilterHelper(unsigned int slot, Long64_t entry, TypeList<ColTypes...>, std::index_sequence<S...>)
{
  return fFilter(fValues[slot][S]->template Get<ColTypes>(entry)...);
}
```

Listing 3.2: Shows two methods which RunAndCheckFilters calls on named filters. Note that RunAndCheckFilters calls CheckFilterHelper implicitly.

the time required to fetch the necessary field values, and the time to compute the operation itself. More formally, we can modify the previous model to the following expression,

$$T_{\text{eventloop}}(N_{\text{events}}) = \sum_{i=1}^{m+1} \left[N_{\text{events}} \cdot f^i_{\text{path}} \cdot \sum_{q \in Q^i} \left[T^{\text{fetch}}_q + T^{\text{compute}}_q \right] \right] \tag{3.1}$$

In this equation, we sum over the latency of each of the $m+1$ execution paths in an RDataFrame program with m filters, to yield the total time spent on executing a single event-loop. The inner summation determines the latency of execution path i, by summing over the latency for each operation in it. More specifically, we sum over each operation q in Q^i where we define the latter as the set of operations in execution path i. Within the inner summation, one can furthermore see that we split the latency of an operation q in the latency to fetch all field values required for it, as well as the latency to compute the operation. By multiplying the time for an execution path i, with the frequency with which the event loop goes in this path, we obtain the latency contribution of the execution path to the event loop as a whole. The frequency of all events which go into execution path i, given by f^i_{path}, multiplied by N_{events}, produces this number.

3.5.4 Latency distributions

3.5.4.1 Determination

Specifically for the LHCB benchmark, we know that `RAction` instances encompass actions and defines. We can thus link the time required to fetch their required field values, and the time to compute these operations themselves, to the time to perform the templated Get function, and the time to perform `fHelper.Exec`, respectively. In the same way, we can link the time to fetch the field values for named filters, as well as the time to perform these filters, to the templated Get function and `fFilter` within `CheckFilterHelper`.

We concretely measure the field value fetching time for all named filters and `RAction` instantiations, by first writing the result of the templated Get function in code listings 3.1 and 3.2 to a separate variable, before passing it to `fFilter` or `fHelper.Exec`. We then measure its duration, by surrounding it with `__rdtsc()` calls. Because we have now separated the field value fetching from the named filter computation as well as the computation required in `RAction` instantiations, we can measure the cycle duration of both `fFilter` and `fHelper.Exec` using `__rdtsc()`. In order to save these durations, we modified the `RAction` and `RFilter` class to contain an unordered map for both the field fetching cycle count as well as the computation cycle count. In each instance of either objects, we thus build the distribution of field value fetching cycle counts and computation cycle counts. We print these distributions at object destruction.

Remember that we evaluate the LHCB benchmark specifically on the B2HHH dataset in RNTuple format without compression. Also recall that we use a c4.8xlarge Amazon EC2 instance running at 2.9 GHZ. Before taking any measurements, we recompiled the ROOT framework to include the instrumentation that builds distributions for `RAction` instances and named filters. We then took 100 measurements of the LHCB benchmark, by running it. Similar as in the previous model calibration, we pin execution threads to cores, and drop operating system caches, before we take any measurement.

3.5.4.2 Preliminary results & Analysis

It turns out that, on average, we get 10.55 seconds (n=100, var=$7.258 \cdot 10^{-4}$) as the sum of the initialization and analysis time. This includes the distribution instrumentation and is very close to the average sum of 10.51 seconds (n=10, var=$4.1 \cdot 10^{-2}$) without instrumentation. The instrumentation latency is thus negligible.

For the LHCB benchmark, each measurement produces one `RAction` distribution, and 9 named filter distributions. This shows that the `RAction` instance encompasses all defines

and the histogram adding operation. By only measuring the field fetching latencies and operation execution latencies as explained before, we hoped to be able to capture the analysis time as in the previous model. Unfortunately, the sum of all histograms is 8.386 seconds on average (n=100, var=$5.784 \cdot 10^{-2}$), only 81% of the average analysis time without instrumentation, whereas the previous model reached 99%. We thus can not ignore the impact of RDataFrame framework functions such as `CheckFilters` when desiring to accurately model the analysis time.

3.5.5 Model calibration & prediction

Being aware of this model limitation, we still wanted to check what latency value a fully calibrated model would produce, for the uncompressed B2HHH dataset. Remember that in each measurement, we get two distributions for each filter. More specifically, a filter field value fetching time distribution, and a filter computation time distribution. Additionally, we have these two distributions for all defines and the histogram adding operations considered together.

After summing each of these distributions across all measurements, we could calculate a mean value representing each distribution. Concretely, at this point we have mean time $T^{\text{compute}}_{\text{filter-}i}$ to compute filter i and a mean time $T^{\text{field-fetch}}_{\text{filter-}i}$ to fetch the field values for this filter. For all defines and the histogram adding operation considered together, we also have a mean time $T^{\text{field-fetch}}_{\text{defines,hist}}$ to fetch the required field values and a mean time $T^{\text{compute}}_{\text{defines,hist}}$ to compute these operations.

Given that the LHCB benchmark first evaluates all 9 filters consecutively, followed by all defines and the histogram adding operation, we have 10 execution paths. For execution path $1 \leq i \leq 9$ we can represent its latency using the previously defined values as $\sum_{j=1}^{i} \left[T^{\text{compute}}_{\text{filter-}i} + T^{\text{field-fetch}}_{\text{filter-}i} \right]$. For the last execution path this is $\sum_{j=1}^{9} \left[T^{\text{compute}}_{\text{filter-}i} + T^{\text{field-fetch}}_{\text{filter-}i} \right] + T^{\text{compute}}_{\text{defines,hist}} + T^{\text{field-fetch}}_{\text{defines,hist}}$. We have now redefined the inner summation in equation 3.1 for the LHCB benchmark.

Before we can use our model to predict a latency, we should still determine the fractions of all events, with which we enter a certain execution path. These are the parameters f^{i}_{path}. When modyfing the LHCB benchmark to include a `Report` call, running the LHCB benchmark on the B2HHH dataset without compression, yields the amount of events entering a certain filter, and the amount of events which pass a filter. We thus know the amount of events which do not pass a filter. Because we also know the total amount of events in the B2HHH dataset, we could calculate the factor of all events which do not pass each filter. These values are the desired f^{i}_{path} values for $1 \leq i \leq 9$. For the last execution path with

$i = 10$, this parameter equals the ratio of events that pass the final filter divided by the total amount of events. Given the found values of f^i_{path}, $T^{\text{compute}}_{\text{filter-}i}$, $T^{\text{field-fetch}}_{\text{filter-}i}$, $T^{\text{compute}}_{\text{defines,hist}}$ and $T^{\text{field-fetch}}_{\text{defines,hist}}$ for $1 \leq i \leq 9$, the model predicts a latency of 8.386 seconds for the B2HHH dataset with 8556118 events. This value is exactly the average sum of all distributions across each measurement, which is impressive.

3.5.6 Analysing the latency distributions

For the `RAction` instantiation, encompassing all defines and the `Histo1D` action, we show its computation and field value fetching latency distributions across all measurements in Figure 3.5. Both distributions have a long tail and predominantly consist of low-valued latencies. We also observe these properties for all named filter distributions, and remember that we also saw this for the latency distribution of `RunAndCheckFilters` in Figure 3.4. We show visually identified computational low-latency areas, for filter 1, filter 6 and filter

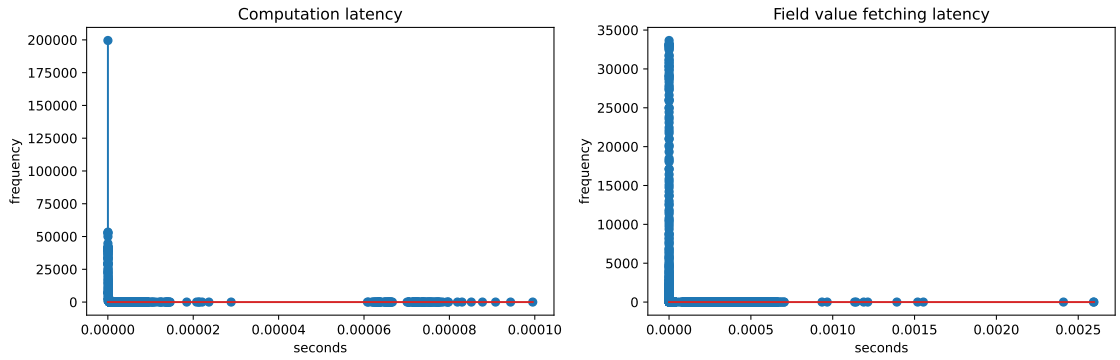


Figure 3.5: Displays the computation and field fetching latency distribution for all defines and the histogram adding operation within the LHCB benchmark applied on the B2HHH dataset without compression, across all 100 measurements on an Amazon c4.8xlarge EC2 instance.

8, in Figure 3.6. These represent the computational low latency areas of all filters. More specifically, the computational low latency area for filter 1 is very similar to that of filters 2-6. Additionally, for filter 6 it is almost identical to that of filter 7. Finally, for filter 8, it is almost the same as for filter 9. For the RAction instantiation, its computational low latency area, is also very similar to the computational low-latency area of filters 8-9.

Contrary to the expectation, we can see that these areas do always not consist of one single identifiable peak. Nevertheless, the mean computational latency for each operation, lies within this low latency area. Within Figure 3.6, we denote the corresponding means using a vertical red line.

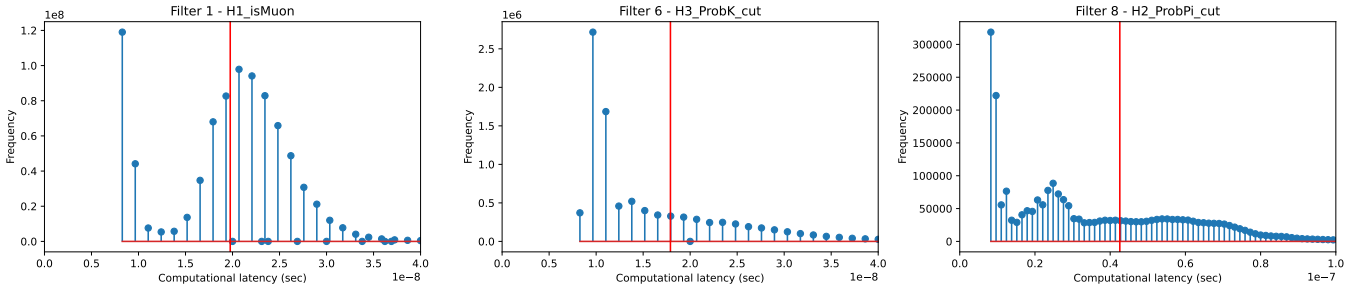


Figure 3.6: Displays the low latency area of the computational latency distributions for filters 1, 6 and 8, within the LHCB benchmark applied on the B2HHH dataset without compression, over 100 measurements on an Amazon EC2 c4.8xlarge instance. Each vertical red line denotes the mean of the distribution. Note that the computational latency distribution corresponds to the latency distribution of fFilter in the RFilter class.

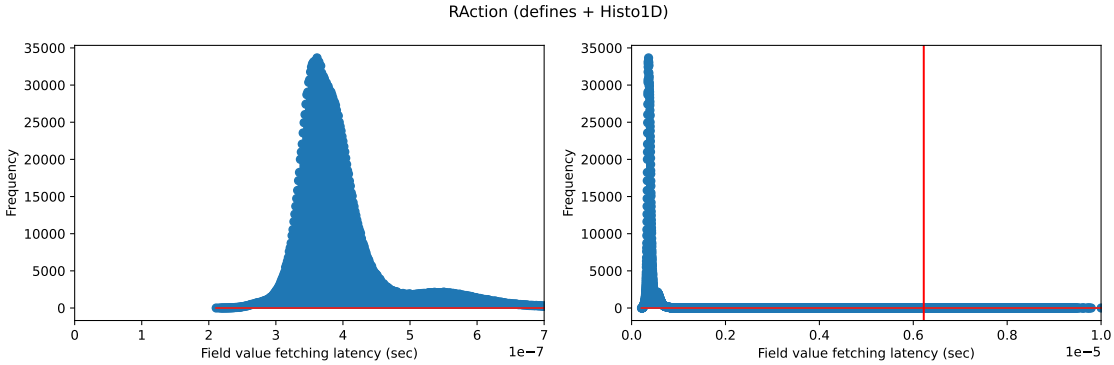


Figure 3.7: Displays the field value fetching latency distribution for the RAction instantiation within the LHCB benchmark applied on the B2HHH dataset without compression for all 100 measurements, at two zoom-levels. We refer to the latency range displayed in the leftmost figure as the low latency field fetching area for the RAction instantiation. In the rightmost figure, the red line shows the distribution mean.

For the field value fetching latency distributions, the low-latency areas also do not always consist of one peak. In these distributions, the mean value sometimes lies greatly outside the low-latency area. More specifically, it sometimes is a factor 8x or more higher than the low latency area upper bound. We show the field value fetching low latency area for the RAction instantiation, as well as the position of the mean with respect to this area, in Figure 3.7. The interested reader can see all other low latency field fetching latency areas for the filters, as well as their corresponding distribution means, in Figure 7.4 of the appendix.

The mean computation latency seems to provide a better summary of its distributions, than the mean field fetching latencies. That is because they always lie within their low

latency areas, whereas the latter do not. It turns out that there is an IO subsystem in RDataFrame applications, which reads the dataset, and causes the field value fetching latency to be non-constant across different requests for data. This is consistent with the observation that the mean field fetching latency does not always seem able to capture its corresponding distributions as well as the mean computation latency.

3.5.7 Next steps

We will study this IO system and modify the model to include its operation. We chose to do so, because the decompression of data, for which we want to investigate the feasibility of GPU offloading, occurs in this system. More precisely, modelling this IO system in more detail, will allow us to understand this decompression and its relation to other components of the RDataFrame framework, which is a prequisite for thinking about possible GPU decompression approaches. It is not important that the model can not predict the analysis time accurately, because we know that the framework overhead causing this, does not relate to decompression.

4

Models incorporating I/O

4.1 IO system operation

4.1.1 RNTuple

The RNTuple format stores a set of C++ object instantiations in a column-wise manner. Given that the term "column-wise" is often used in the context of tabular data, this might seem like a counter-intuitive combination with C++ objects. It is therefore important to know, that these C++ objects are, at least conceptually speaking, mapped to table rows prior to storage. Columns of this table store values of a specific object field. Within RNTuple, objects or rows are referred to as events, whereby a range of consecutive rows is called an event range.

Even though being described as simply column-wise, storage is actually more complicated, and utilises the notion of clusters and pages. Clusters store an event range, whereby storage is only column-wise within those clusters. In such a cluster, each column is further segmented in pages. These pages must thus store values of a single field.

Apart from storing C++ objects, the RNTuple format also reserves some space for meta-data. For a stored C++ class, the meta-data that is accessible describes its fields, associated columns and their relationships(58). Clusters are also described, through their size and location. Finally, columns within a cluster are described through their element index range, a list of pages that contain these elements, and page locations.

We will now elaborate on the mapping between C++ object fields and on-disk columns. The complexity of this mapping is dependent on the field type. Simple object fields such as integer or float struct members, are directly mapped to an on-disk column of the corresponding type. A more complex type such as a std::vector<int>, is mapped to an index column and a value column of type int. The index column denotes the sections of the

value column that corresponds to the std::vector<int> of each event. Note that the value column indexing is cluster-local. Also note that the storage of an individual collection, does not cross cluster boundaries. This also follows from the earlier remark that clusters store an event range fully.

More concretely, the index column contains an entry for each event. The std::vector<int> of event i is then stored in the following range of the value column,

$$\begin{cases} 0 \leq j < \text{index\_col}[i] & \text{, if } i == 0 \\ \text{index\_col}[i-1] \leq j < \text{index\_col}[i] & \text{, else} \end{cases}$$

4.1.2 RClusterPool

4.1.2.1 Reading thread

The reading thread of RClusterPool executes its ExecReadClusters(59) method. This thread reads clusters in batches. This does not mean that the thread reads each cluster page. At least all cluster pages are read so that the requested columns are available (60).

Read requests enter in a queue, and are wrapped in a RReadItem (61) object that consists of a bunch id (indicating the read batch), a cluster id and a std::promise to a RCluster reference. Note that the RCluster reference promise is set only, after a fulfilled read request has passed the unzip thread. The reading thread performs pending read requests in its queue, that belong to the same batch, trough the LoadClusters method (60) on its page source. Between those batches, it potentially deletes some read requests that are not required anymore. Note that a page source is an interface to read from an NTuple file, and its meta-data(62). It is initialized by a subset of physical columns which it potentially maps to alias columns.

Read requests can become obsolete because they are not necessarily a direct consequence of data requirements. More specifically, when reading a cluster is required at some point, read requests for nearby clusters are also posted. When the access pattern with respect to the clusters is not sequential, some read requests might not be required anymore.

A fulfilled cluster read request results in a RCluster object, that represents its in-memory potentially packed and compressed pages using ROnDiskPage (63) objects. The pages themselves can be memory mapped or a priori copied in a buffer(64). From now on we will refer to packed and compressed pages as sealed pages.

Resulting RCluster objects are passed to the unzip thread, by placing them in its in-queue, after being wrapped in an RUnzipItem. Apart from containing an RCluster reference, an RUnzipItem (65) object must also contains the promise that was in the corre-

sponding RReadItem. The latter is to enable setting this promise after the fulfilled read request has passed the unzip thread.

4.1.2.2 Unzip thread

As implied by its name and the code in ExecUnzipClusters (66), the unzip thread should unseal the pages of each cluster in its incoming queue. Given that unseal jobs are actually performed in a task scheduler, it is prerequisite that this scheduler is set (67). In the context of the current RDataFrame implementation, this is not the case, however [1]. Within this context, the unzip thread simply loops over all incoming RUnzipItem objects, setting the RCluster promise to its RCluster immediately.

The unzip thread actually calls the UnzipCluster method of its page source, for every queued to-be unsealed cluster. There are three concrete page source implementations: RPageSourceFile (68), RPageSourceDaos (69), and RPageSourceFriends (70). The meaning of the RPageSourceDaos implementation might be clarified by the meaning of an DAOS container, which is an abstract object storage system(71). RPageSourceFriends serves as a page source when data columns actually originate from different sources. [2] Within the RDataFrame context, RPageSourceFriend is used when some so-called friend columns are defined at runtime, and are used for filtering, together with columns originating from another source.

Whenever a task scheduler is set, the page source performs the requested unsealing, if necessary, within the UnzipClusterImpl (72) method. This function loops over all NTuple columns and each page in each column, based on NTuple meta-data. For each page, a task is created that is added to the scheduler. When executed, such a task fetches an ROnDiskPage from an RCluster, which it converts to an RSealedPage, after which the UnSealPage method of the page source unseals the page, if necessary (73).

Within this method, two buffers are potentially allocated for the results of unpacking and unzipping, respectively. The task wraps the single result buffer in a RPage object (74) which also stores some meta-data[3]. Such an RPage is then added to a page pool, a page source member which acts as a cache of column pages(75).

After adding all unsealing tasks to the task scheduler, the UnzipClusterImpl call blocks until all tasks have been completed.

[1] According to Jakob Blomer

[2] According to Jakob Blomer

[3] The amount of elements in the page, the size of such an element, the first column index stored in this page, a cluster id, and the starting index of an element in a column of this cluster.

4.1.2.3 GetCluster

The GetCluster method(76) is called with a requested cluster id, and a set of physical columns (all non-alias columns). This method then sets up $2 \cdot$ fClusterBunchSize cluster load requests, sequentially, starting from the requested cluster, for the set of columns, separated in two batches. In the following, we will continue to describe the inner workings of this method.

RClusterPool contains a buffer fPool of RCluster objects which have previously passed the IO/unzip pipeline. After the creation of the set of load requests, all buffered RCluster objects not in this set, are removed. Apart from a buffer of passed RCluster objects, RClusterPool also contains a list of in-flight load operations with RInFlightCluster objects.

Whenever an in-flight load operation is not in the current set of to-be satisfied load requests, it is marked for cancellation. It could also occur that an in-flight load operation (partially) satisfies a current load request. A current cluster load request X is partially satisfied by an in-flight load operation Y when Y loads a subset of the requested cluster columns X. In either case, the to-be satisfied load request is modified so that the same data is not loaded twice. Completed load requests result in a RCluster object that is added to or merged with an existing RCluster object, within fPool. Merging two RCluster objects refers to placing the ROnDiskPage references of one cluster into the other. Obviously, the load request is also removed from the in-flight list.

Newly added clusters to fPool, as well as surviving clusters added in previous invocations, could (partially) satisfy some of the to-be-satisfied load requests. Therefore we delete or strip load requests from the to-be-satisfied set by evaluating the RCluster objects already cached in fPool.

What remains are all load requests that are not cached and not currently being processed by in-flight operations. These load requests are initiated by placing them in the list of in-flight operations, as well as the IO thread queue, in two cases. That is first of all, when there are more remaining load requests than fClusterBunchSize. This is also the case, when there are less load requests, on the condition that it contains the cluster on which the function was called, or the last load request of the initially created second batch.

GetCluster returns when the requested cluster and its columns are available. This can be immediately, when the cluster is already in fPool. When the requested cluster columns are not fully available, the function waits for this to occur. It then returns, after adding the requested cluster columns to fPool, additionally removing the request from the in-flight list.

4.2 Improved single-threaded model

Now that we have a clear picture of IO system operation within RDataFrame programs, we will use this knowledge to create a more accurate latency model for them, in a single-threaded context. Remember that we constrain our focus to RDataFrame programs with a single event-loop, wherein each iteration proceeds through a chain of transformations and actions. Additionally, note that in building this improved model, we limit the considered transformations to filters and defines.

4.2.1 Modelling the computational latency

Lets start by modelling the pure computational latency contribution of all filters, defines and actions, in a way strongly resembling the approach used in formulating our first models. In an RDataFrame program with m filters, there are $m + 1$ execution paths, and for each execution path we assume a constant computational latency T_j^{comp}. Implicitly, this makes the assumption that the computational latency for each filter, define and action is constant as well. When additionally denoting the relative occurrence of the amount of events that follow execution path j as f_j^{comp}, we can formulate the total computational latency within a single-threaded RDataFrame program as,

$$T_{\text{comp}}(N) = N \cdot \sum_{j=1}^{m+1} f_j^{\text{comp}} T_j^{\text{comp}}$$

There are nine filters within the LHCB benchmark, so then $m = 9$.

4.2.2 Fetching field values

Each filter, define and action might require multiple on-disk field values before they can be computed. A read call is thus invoked on each required RFieldBase object. If the call can not be satisfied by a cache storing the latest retrieved value, deserialization is started. Deserialization is a process that constructs the requested value, by retrieving values from one or more physical columns which are represented as RColumn objects.

Retrieving such a physical column value is yet another call, which might be satisfied by a unary sized unsealed page cache owned by the column. If this is not the case, the required unsealed page is retrieved from the underlying page source. There is usually just a single page source. In the case of the LHCB RDataFrame benchmark, that is a RPageSourceFile object, which reads from an RNTuple file. Note that the RPageSourceFile can also read from other file formats such as the TTree format.

This page source contains a cache of $2 \cdot$ fClusterBunchSize unsealed pages, which might satisfy the request. It additionally caches the latest fetched cluster from the RClusterPool. If both caches do not hold the requested page, a fetch request is initiated on the RClusterPool.

A fetch request is actually a getCluster call which has been described in detail before. Such a call updates the current RClusterPool window to the current required cluster as well as the next $2 \cdot$ fClusterBunchSize $- 1$ clusters. An RClusterPool instantiation continually attempts to satisfy its window of $2 \cdot$fClusterBunchSize clusters. Resulting clusters belonging to the current window are cached. Read requests themselves are satisfied in a separate thread, in batches of fClusterBunchSize clusters. Whenever the cluster requested by the page source is ready, the cluster cache is updated, the requested page is unsealed, added to the page source page cache and the column-local cache. Finally, the requested value is returned back to the field.

Now one can clearly see that our focus on single event-loop RDataFrame applications reduces modelling effort, because we do not have to consider data reuse through caches, between different event-loops.

4.2.3 Modelling the full latency

We will now incrementally model the full latency $T(N)$ of a single-threaded RDataFrame program with a single event-loop which can only contain filters, defines and actions. In addition to the computational latency model given before, $T(N)$ also incorporates the IO, unsealing and deserialization time. We start model creation by describing the first field fetch occurring in the first filter.

First field value fetch As for any field fetch, the first field fetch will trigger loads on the columns that make it up, in order to construct a value in the deserialization process. The first column load request triggers the RClusterPool to load the first batch containing the cluster with the requested page. Only after reading this batch has completed, the page source will be able unseal and serve all page requests placed by the columns.

Given that we should wait on reading the first batch, and for the unsealing of all required column pages to construct the first field value, we add corresponding terms to the latency model shown below. The unsealing time for all used column pages, is captured with the summation term. Essentially, we sum the unsealing latency of each used page for the first field value, per column owned by the field. For a column owned by the first field, the unsealing latency for the first value is determined by multiplying the amount of used pages

for the first value N^{firstval} with the average unsealing time for a page of a column of type t using a compression algorithm A and a compression level L i.e. $T_{\text{unseal}}^{\text{page-}(t,A,L)}$. We will later elaborate on the variables over which this average is taken. Previously we stated that we sum over all used columns used by the first field. To be more exact, we actually sum over elements in C_1, which is a set of 4-tuples $(N^{\text{firstval}}, t, A, L)$ for each column owned by the first field. Due to this construction, A and L can only vary on a column-basis. This is a modelling simplification given that these can actually be varied across different clusters. The final term in the equation $T_{\text{deser}}^{\text{field-1}}$ represents the latency for constructing the first field value, which we call deserialization. Yet-to be added terms are represented by the dots.

Within this first field fetch, we do not represent the time that is spent in interacting with the caches and the getCluster call itself. We assume that this is negligible with respect to $T(N)$. The same assumption will be made in to-be formulated terms of this model, even though this is then less likely to hold up, due to its presence in terms dependent on N.

$$T(N) = T_{\text{readbatch}}^{\text{first}} + \sum_{\left(N^{\text{firstval}},t,A,L\right)\in C_1} \left[N^{\text{firstval}} T_{\text{unseal}}^{\text{page-}(t,A,L)}\right] + T_{\text{deser}}^{\text{field-1}} \dots$$

The previous discussion characterised $T_{\text{unseal}}^{\text{page-}(t,A,L)}$ as an average. This is due to the fact that the actual page decompression time is dependent on the amount of elements stored within the page, as well as the data-dependent realised compression ratio.

One of the reasons why the amount of stored elements in a page varies, is due to the way they are filled. Pages are filled on a cluster basis. That means that all pages in a cluster are first allocated, before they are filled with data. Each page of a certain type has an uncompressed target size, and clusters have a maximum size too. It could occur that the maximum cluster size is reached before pages reach their uncompressed target size, thus making the amount of stored elements within a page smaller then the target size dictates. Given that the target size of a page type does not likely divide an uncompressed column evenly, the last page of a certain column type will also not be fully saturated. More specifically, if this last page is smaller than 50% of the target size for this type, it will be merged with the previous page. The last page of each cluster column will thus range between 50% and 150% of the target size.

With this info, we redefine $T_{\text{unseal}}^{\text{page-}(t,A,L)}$ as the average latency to unseal a cluster column page of type t with compression algorithm A and compression level L, over the possible amount of elements stored in such a page, as well as the different occurring data-contents of such a page. The latter leads to the various compression ratios.

Computational, deserialization & unsealing latency Obviously, the computational latency should also be a part of $T(N)$,

$$T(N) = T^{\text{first}}_{\text{readbatch}} + \sum_{\left(N^{\text{firstval}},t,A,L\right)\in C_1} \left[N^{\text{firstval}} T^{\text{page-}(t,A,L)}_{\text{unseal}}\right] + T^{\text{field-1}}_{\text{deser}} + T_{\text{comp}}(N) \dots$$

In addition to this, we should also add latency terms for fetching field values for each filter, define and action, apart from the first field value fetch. Given the previously described assumption, we represent this using deserialization and unsealing latency only.

We will first add the deserialization latency to the model. The amount of deserialized fields depends on the followed execution path. For example, when filter 2 is rejected, only field values for the first two filters need to be deserialized. We can thus add a term similar to $T_{\text{comp}}(N)$ where we replace the latency of a execution path, by the latency to deserialize all field values that are required for this path. The latency to deserialize all field values for execution path j is given by the inner summation, which loops over all unique fields in set A_j. This set is defined for all $1 \leq j \leq m+1$ and contains the amount of unique fields occurring in path j . Note that it only contains unique fields because deserialization only needs to occur once per field value in an event loop iteration. In other words, when a field value is used multiple times, the unary-sized cache in the corresponding RField instantiation is used, instead of repeating the process. Also note that the summation covers $T^{\text{field-1}}_{\text{deser}}$ so it is not written as a separate term anymore.

$$T(N) = T^{\text{first}}_{\text{readbatch}} + \sum_{\left(N^{\text{firstval}},t,A,L\right)\in C_1} \left[N^{\text{firstval}} T^{\text{page-}(t,A,L)}_{\text{unseal}}\right] + \sum_{j=1}^{m+1} \left[N f_j^{\text{comp}} \sum_{f\in A_j} \left[T^{\text{field-}f}_{\text{deser}}\right]\right] + T_{\text{comp}}(N) \dots$$

We will now add a term to the model that represents the required unsealing latency for each field value fetch. Unsealing does not occur for each field value fetch, but rather for each used page within each used column. Our access pattern with respect to events is sequential. Therefore, values of a column owned by a field, are also accessed sequentially. Given the column-local unary-sized page cache, it would seem like repeated unsealing of the same page does not occur.

Albeit unlikely, this can occur due to the fact that columns may be referenced more than once in a single field, or across different fields. Note that reuse of the same field in the same event loop iteration, will use the single deserialized value cache, and thus prevent duplicate unsealing.

An example scenario in which repeated unsealing occurs, is the following. Assume that a column is referenced in fields F1 and F2, and is used at different page locations in the

same event loop iteration. At the end of the event loop iteration, the column will hold the unsealed page used to construct a F2 field value. When enough pages are unsealed between the construction of a value of F1 and F2, the unsealed page for F1 will not be in the page source cache anymore. In the next event loop iteration, the same page must therefore be unsealed again. Obviously, this assumes that the next value of F1 is required, and that this requires values that belong to the same page as in the previous iteration.

Regardless of the amount of unseals of each page, the total latency contribution of unsealing can be obtained, by summing the time spent on unsealing each used column. Defining the set of columns owned by all unique fields f as F_f with $1 \leq f \leq F$, $\bigcup_{f=1}^{F} F_f$ gives a set of all columns used in the RDataFrame program, over which we can sum.

The second term in the updated model shown below, sums over each column q in this set. For each q, the unsealing latency contribution is formalised with the inner fraction.

More specifically, the numerator quantifies the amount of unsealed bytes that must be produced for column q. This value is calculated from S_q and f_q^{unseal}, where S_q denotes the total unsealed byte size of column q whereas f_q^{unseal} quantifies the percentage of S_q which is actually produced by unsealing. f_q^{unseal} can capture the case in which a subset of all pages of q are unsealed, as well as the rare case of duplicate page unsealing. The case in which some pages need not to be unsealed, is rare as well given the filtering strength that is required for this to occur. It is therefore likely that $f_q^{\text{unseal}} = 1$. Whereas we do not make this assumption, we do assume that f_q^{unseal} is constant and thus independent of N. Obviously, this parameter depends on the data-set type that is filtered, as well as the filter code itself.

The denominator quantifies the speed at which unsealed bytes for pages of column q are produced. Obviously, dividing the amount of produced unsealed bytes by this speed, yields the time spent unsealing column q. We assume that there is a separate but constant unsealing speed for each possible triplet consisting of a column type, a compression algorithm and a compression level. That is why the superscript of λ contains operators that return these variables for a column q, in the order that has been given before.

An observant reader might have noticed that the term describing the latency contribution of unsealing the first field value, is not included anymore. That is because it is already captured by this summation.

$$T(N_{\text{events}}, S) =$$
$$T^{\text{first}}_{\text{readbatch}} + \sum_{q \in \bigcup_{f=1}^{F} F_f} \left[\frac{S_q \cdot f_q^{\text{unseal}}}{\lambda^{t(q),C(q),L(q)}} \right] + \sum_{j=1}^{m+1} \left[N_{\text{events}} f_j^{\text{comp}} \sum_{f \in A_j} \left[T^{\text{field-}f}_{\text{deser}} \right] \right] +$$
$$T_{\text{comp}}(N_{\text{events}}) \dots$$

The current latency characterisation assumes that pages are always immediately available for unsealing, deserialization and computation within filters, defines or actions. In reality, reading of batches within the read thread, overlaps with this. When the read thread latency dominates unsealing, deserialization and other computational latency, reading of batches occurs continuously. By additionally assuming that the first and other batches are read with a fixed latency $T^{\text{first}}_{\text{readbatch}}$ and $T^{\text{non-first}}_{\text{readbatch}}$, respectively, this case has latency,

$$T^{\text{first}}_{\text{readbatch}} + \max((N_{\text{clusters}}/N_{\text{batch}}) - 1, 0) \cdot T^{\text{non-first}}_{\text{readbatch}}$$

where N_{batch} is the amount of clusters in a batch. We chose to model the batch reading latency with an initial latency and an non-initial latency, to take into account the common IO throughput ramp-up. In reality, the batch reading latency is time-dependent, especially in an environment where IO resources are shared.

Combining the IO dominating latency model with the previous characterisation of $T(N_{\text{events}}, S)$ using the max operator, allows us to model both cases at once,

$$T(N_{\text{events}}, S, N_{\text{clusters}}) = T^{\text{first}}_{\text{readbatch}} + \max(T_{\text{unseal}}(S) + T_{\text{deser}}(N_{\text{events}}) + T_{\text{comp}}(N_{\text{events}}), T_{\text{IO}}(N_{\text{clusters}}))$$
$$T_{\text{unseal}}(S) = \sum_{q \in \bigcup_{f=1}^{F} F_f} \left[\frac{S_q \cdot f_q^{\text{unseal}}}{\lambda^{t(q),C(q),L(q)}} \right]$$
$$T_{\text{comp}}(N_{\text{events}}) = \sum_{j=1}^{m+1} N_{\text{events}} f_j^{\text{comp}} T_j^{\text{comp}}$$
$$T_{\text{deser}}(N_{\text{events}}) = \sum_{j=1}^{m+1} \left[N_{\text{events}} f_j^{\text{comp}} \sum_{f \in A_j} \left[T^{\text{field-}f}_{\text{deser}} \right] \right]$$
$$T_{\text{IO}}(N_{\text{clusters}} - 1) = \max\left(\frac{N_{\text{clusters}}}{N_{\text{batch}}} - 1, 0 \right) \cdot T^{\text{non-first}}_{\text{readbatch}}$$

4.2.3.1 LHCB benchmark simplifcations

In the LHCB benchmark, only simple double and integer fields are used, in which values of a single column are directly mappable to field values. Each field thus owns a single

column. In other words F_f is of unary size for $1 \leq f \leq F$. Across all unique fields, there may also not be duplicate columns. If two unique fields could own the same column, they would namely no longer be unique. Thus $\forall i \neq j \in 1 \leq f \leq F, F_i \cap F_j = \emptyset$. Due to these two observations, we can simplify the summation in T_{unseal} as follows,

$$T_{\text{unseal}}(S) = \sum_{f=1}^{F} \left[\frac{S_{F_f} \cdot f_{F_f}^{\text{unseal}}}{\lambda^{t(F_f),C(F_f),L(F_f)}} \right]$$

where F_f should be read as referring to the single column that is in that set.

We can further simplify this equation, by noting that the compression level and algorithm are fixed on a global level, and that there are 15 fields that use a column of the type `SplitReal64` and 3 fields that use a column of the type `SplitInt32`. This info was derived by running the `ntupleinfo` program from the `iotools` repository (77) on the LHCB dataset.

Each column element has a fixed size, dependent on its type. Given that there are as much events N_{events} as there are field values of each type, where each field value directly maps to a column value, we can write S_{F_f} in terms of N_{events} and the size of a column element. The `RNTupleDescriptorFmt` (78) class is invoked in `ntupleinfo` to print info of all columns in a dataset. This class was modified to also print the element size. Repeating execution of the `ntupleinfo` program on the LHCB dataset then yields the required element sizes, with which we can write S_{F_f} as $S_{F_f} = N_{\text{evens}} \cdot 4$ bytes if $t(F_f) = \text{SplitInt32}$ and $S_{F_f} = N_{\text{events}} \cdot 8$ if $t(F_f) = \text{SplitReal64}$. All in all, we can thus write $T_{\text{unseal}}(S)$ as follows,

$$T_{\text{unseal}}(N_{\text{events}}) = 15 \cdot \frac{N_{\text{events}} \cdot 8}{\lambda^{\text{SplitReal64},C,L}} + 3 \cdot \frac{N_{\text{events}} \cdot 4}{\lambda^{\text{SplitInt32},C,L}}$$

Additionally, the construction of field values i.e. deserialization is not necessary, as values embedded in read pages can directly be used. The total equation can thus be rewritten as follows for the LHCB benchmark,

$$T(N_{\text{events}}, N_{\text{clusters}}) = T_{\text{readbatch}} + \max(T_{\text{unseal}}(N_{\text{events}}) + T_{\text{comp}}(N_{\text{events}}), T_{\text{IO}}(N_{\text{clusters}}))$$

4.3 Multi-threaded case

Event-loops in an RDataFrame program can be performed using multiple threads, by enabling implicit multi-threading (35). These threads can then perform all event processing operations in parallel, except for reading which is still performed in a single IO thread. Before an event loop is started, the range of all to-be-processed events is first of all split into as many parts as there are threads, whereby the last thread gets the remainder (79, 80).

Given that each thread performs the same operations for computing actions, filters and defines, as well as for deserialization, but for a subset of events, we can model the computational latency and the deserialization latency for thread $1 \leq t \leq N_{\text{threads}}$, as follows. This assumes that the difference between the amount of events assigned to the last thread, and all other threads, is negligible. In other words, $N_{\text{events}} \% N_{\text{threads}}$ must be very small with respect to $\lfloor N_{\text{events}}/N_{\text{threads}} \rfloor$.

$$T'_{\text{comp}}(N_{\text{events}}) = T_{\text{comp}}(N_{\text{events}}/N_{\text{threads}}) = \frac{1}{N_{\text{threads}}} T_{\text{comp}}(N_{\text{events}})$$

$$T'_{\text{deser}}(N_{\text{events}}) = T_{\text{deser}}(N_{\text{events}}/N_{\text{threads}}) = \frac{1}{N_{\text{threads}}} T_{\text{deser}}(N_{\text{events}})$$

For each used column q in the RDataFrame program with an unsealed size S_q, each thread t gets a partition of size S_q^t, which contains all required column pages to process the assigned events. As in the single-threaded model, a thread might produce unsealed data in its partition more than once, or not at all. This is due to unnecessary pages or pages which are unsealed more than once. All in all, a thread t thus can thus be stated to produce unsealed data of size,

$$\sum_{q \in \bigcup_{f=1}^{F} F_f} S_q^t \cdot f_q^t$$

In the single-threaded model we assumed that the produced fraction f_q of S_q, is independent of the data-set size, for a data-set of a certain type. Given that the partition assigned to a thread has the same type as the original unpartitioned dataset, f_q^t should be independent from the respective thread. We can thus rewrite the equation governing the amount of uncompressed data produced by thread t as,

$$\sum_{q \in \bigcup_{f=1}^{F} F_f} S_q^t \cdot f_q$$

Dividing by the speed at which unsealed data is produced, then yields the time that thread t spends unsealing its data,

$$\sum_{q \in \bigcup_{f=1}^{F} F_f} \frac{S_q^t \cdot f_q}{\lambda^{t(q),C(q),L(q)}}$$

When furthermore assuming that the partitions of S_q assigned to all threads are of equal size, which means $S_q^t = S_q / N_{\text{threads}}$, the time spent unsealing is the same for each thread,

$$T_{\text{unseal}}^{\text{multithread}}(S) = \frac{1}{N_{\text{threads}}} \sum_{q \in \bigcup_{f=1}^{F} F_f} \frac{S_q \cdot f_q}{\lambda^{t(q),C(q),L(q)}} = \frac{1}{N_{\text{threads}}} T_{\text{unseal}}(S)$$

4.4 CPU decompression models

Up till now we have created a latency model for single-threaded and multi-threaded unsealing in RDataFrame programs. In the context of creating and quantifying the effect of various GPU decompression offloading approaches, we want to model the decompression latency in isolation. Remember that page unsealing consists of page decompression, which is sometimes followed by page unpacking. In order to model the total decompression latency in RDataFrame applications, we will simply reuse both unsealing models, slightly changing the meaning of some terms.

$$T_{\text{decompress}}^{\text{singlethread}}(S) = \sum_{q \in \bigcup_{f=1}^{F} F_f} \left\lceil \frac{S_q \cdot f_q^{\text{decompress}}}{\lambda^{t(q),C(q),L(q)}} \right\rceil \tag{4.1}$$

$$T_{\text{decompress}}^{\text{multithread}}(S) = \frac{1}{N_{\text{threads}}} T_{\text{decompress}}(S) \tag{4.2}$$

More specifically, within Equation 4.1 S_q now refers to the amount of produced decompressed data for an used column q. Additionally, $\lambda^{t(q),C(q),L(q)}$ will now refer to the speed at which a single CPU-thread can produce uncompressed data, for a certain dataset type, after the application of a compression algorithm/level. In the same way, we can also reuse the unsealing latency equation that we tailored for the LHCB benchmark and its dataset type.

$$T_{\text{decompress}}^{\text{LHCB,B2HHH}}(N_{\text{events}}) = 15 \cdot \frac{N_{\text{events}} \cdot 8}{\lambda^{\text{SplitReal64},C,L}} + 3 \cdot \frac{N_{\text{events}} \cdot 4}{\lambda^{\text{SplitInt32},C,L}} \tag{4.3}$$

We will now move onto describing the calibration/verification process for the single-threaded and multi-threaded decompression latency model, specifically for the LHCB application and its B2HHH dataset.

4.4.1 Single-threaded model calibration/verification

Work in progress.

4.4.2 Multi-threaded model calibration/verification

Work in progress.

5

GPU offloading

Currently, the CPU performs each page decompression. Instead of this, we can also move compressed pages to the GPU, perform the decompression there, and then move them back to the CPU. We can describe the latency of any GPU offloading implementation that performs these operations sequentially, with the following equation.

$$T_{\text{decompress}}^{\text{GPU-simple}} = T_{\text{H2D}} + T_{\text{decompress}}^{\text{GPU}} + T_{\text{D2H}} \tag{5.1}$$

Note that the abbreviations H2D and D2H stand for "host to device" and "device to host", respectively. We refer to the CPU as the host, and the GPU as the device.

Throughout this chapter, we will start by thinking about offloading decompression to the GPU from a simple perspective, striving to make the implementation progressively more complicated until we predict an overall performance improvement. Apart from minimising the implementation effort, this will also not unnecessarily complicate the performance modelling. On a more practical note, we will compare our offloading approaches specifically for NVIDIA GPUs, by calibrating models for each approach, specifically on these GPUs. The ideas corresponding to each GPU offloading approach, are nevertheless portable.

5.1 Unary page transfer and decompression

The simplest GPU offloading approach, takes the single-threaded CPU implementation as a starting point. This implementation decompresses each page sequentially on the CPU. We can replace the routine of decompressing an individual page, by moving it to the GPU, decompressing it there and then finally moving it back. This is the main idea underlying our first offloading approach.

In order to provide NVIDIA GPUs with work, the concept of CUDA streams comes into play. A CUDA stream resembles a queue, in which one can place computational and data transfer tasks. GPUs can feed off multiple streams, which allows task overlapping, leading to better utilisation. Given our single-threaded CPU implementation, we can only use one stream, however.

Our first offloading implementation will thus consist of one CUDA stream, that we will fill with page transfer tasks, as well as page decompression tasks. By creating, calibrating and making predictions using a latency model for the sum of time spent on transferring individual pages, we will investigate whether this method delivers sufficient performance.

5.1.1 Modelling the host to device transfer time

T_{H2D} and T_{D2H} represent the sum of the transfer time of all used pages in a data-set to and from the GPU, respectively. We start by writing an expression for T_{H2D}, that describes transferral of all used compressed pages to the GPU. One obtains this value by summing the transferral time T^q_{H2D} of all used pages of each used column q. We define the set of all columns that an RDataFrame program uses as $\mathcal{F} = \cup_{f=1}^{F} F_f$. Remember that an RDataFrame program uses F fields, whereby the set F_f describes all columns that a field f requires to construct its values. Taking the union of all these sets thus yields the set of all columns that an RDataFrame program uses.

$$T_{\text{H2D}} = \sum_{q \in \mathcal{F}} T^q_{\text{H2D}} \tag{5.2}$$

Introducing the compressed total size S_q^{compress} of a column q in this equation, and the size factor of this compressed column f_q^{compress} which needs decompression, yields the following with the help of some algebra. Note that f_q^{compress} can be bigger than one due to duplicate page decompression, or smaller than one due to page decompression skips. It is most likely

one due to the rarity of these events.

$$\begin{aligned}
\sum_{q\in\mathcal{F}} T^q_{\mathrm{H2D}} &= \sum_{q\in\mathcal{F}} \frac{S_q^{\mathrm{compress}} \cdot f_q^{\mathrm{compress}}}{S_q^{\mathrm{compress}} \cdot f_q^{\mathrm{compress}}} \cdot T^q_{\mathrm{H2D}} \\
&= \sum_{q\in\mathcal{F}} S_q^{\mathrm{compress}} \cdot f_q^{\mathrm{compress}} \cdot \frac{T^q_{\mathrm{H2D}}}{S_q^{\mathrm{compress}} \cdot f_q^{\mathrm{compress}}} \\
&= \sum_{q\in\mathcal{F}} S_q^{\mathrm{compress}} \cdot f_q^{\mathrm{compress}} \cdot \frac{1}{\frac{S_q^{\mathrm{compress}} \cdot f_q^{\mathrm{compress}}}{T^q_{\mathrm{H2D}}}} \\
&= \sum_{q\in\mathcal{F}} S_q^{\mathrm{compress}} \cdot f_q^{\mathrm{compress}} \cdot \frac{1}{\lambda_q^{\mathrm{H2D}}} \\
&= \sum_{q\in\mathcal{F}} \frac{S_q^{\mathrm{compress}} \cdot f_q^{\mathrm{compress}}}{\lambda_q^{\mathrm{H2D}}}
\end{aligned}$$

It is obvious that S_q^{compress} depends on the amount of data-set entries. Additionally, we can rewrite it in terms $S_q^{\mathrm{uncompress}}$, the uncompressed total size of a column q , with the help of the compression ratio r_q of that column. We assume that each r_q is independent of the size of a data-set of a certain type. It is thus a calibratable parameter for a specific data-set type.

$$\begin{aligned}
T_{\mathrm{H2D}} &= \sum_{q\in\mathcal{F}} \frac{S_q^{\mathrm{compress}} \cdot f_q^{\mathrm{compress}}}{\lambda_q^{\mathrm{H2D}}} \\
&= \sum_{q\in\mathcal{F}} \frac{\frac{S_q^{\mathrm{uncompress}}}{r_q} \cdot f_q^{\mathrm{compress}}}{\lambda_q^{\mathrm{H2D}}}
\end{aligned} \tag{5.3}$$

One can also write the numerator in terms of $f_q^{\mathrm{uncompress}}$, the factor of all uncompressed bytes in column q that decompression needs to produce. In other words,

$$\frac{S_q^{\mathrm{uncompress}}}{r_q} \cdot f_q^{\mathrm{compress}} = \frac{S_q^{\mathrm{uncompress}} \cdot f_q^{\mathrm{uncompress}}}{r_q}$$

This implies that f_q^{compress} and $f_q^{\mathrm{uncompress}}$ are identical. Defining f_q as either of these constants, allows us to rewrite T_{H2D} once more,

$$T_{\mathrm{H2D}} = \sum_{q\in\mathcal{F}} \frac{\frac{S_q^{\mathrm{uncompress}}}{r_q} \cdot f_q}{\lambda_q^{\mathrm{H2D}}} \tag{5.4}$$

We will now further study the λ_q^{H2D} term. First of all we rewrite it in terms of its definition,

$$\lambda_q^{\mathrm{H2D}} = \frac{S_q^{\mathrm{compress}} \cdot f_q}{T^q_{\mathrm{H2D}}} \tag{5.5}$$

We can write T^q_{H2D} as the sum of the transfer time spent on each compressed page size occurring in the column. The time spent on a single compressed page size is the product of the latency for transferring this size with $N_q^{\text{decompress-compressed-pages-i}}$, the amount of decompressions that occur for a compressed page of a size indexed with i within column q. We require this product to account for the separate compressed page transfers to the GPU, which we perform for each necessary decompression. By applying the $T^{\text{size}}_{\text{H2D}}$ function on a compressed page of a certain size, we get the latency for transferring this page from the host to the device. We will determine this function experimentally.

$$\lambda_q^{\text{H2D}} = \frac{S_q^{\text{compress}} \cdot f_q}{\sum_i N_q^{\text{decompress-compressed-pages-i}} \cdot T^{\text{size}}_{\text{H2D}}\left(S_{q,i}^{\text{page-compress}}\right)} \tag{5.6}$$

The numerator within this expression gives the total amount of compressed bytes of column q which the implementation would need to transfer to the GPU. We can express the numerator in terms also occurring in the denominator.

$$\lambda_q^{\text{H2D}} = \frac{\sum_i N_q^{\text{decompress-compressed-pages-i}} \cdot S_{q,i}^{\text{page-compress}}}{\sum_i N_q^{\text{decompress-compressed-pages-i}} \cdot T^{\text{size}}_{\text{H2D}}\left(S_{q,i}^{\text{page-compress}}\right)} \tag{5.7}$$

Within a column q, for a size indexed by i, one can rewrite $N_q^{\text{decompress-compressed-pages-i}}$ as the product of the amount of compressed pages of that size multiplied by a factor $f_q^{\text{decompress-compressed-pages-i}}$ that can capture duplicate or the skipping of page decompressions. Finally, we can then rewrite the amount of compressed pages of a size indexed by i as a fraction $f_q^{\text{compressed-pages-i}}$ of the total amount of pages N_q^{pages} within this column. All in all,

$$\begin{aligned} N_q^{\text{decompress-compressed-pages-i}} &= N_q^{\text{compressed-pages-i}} \cdot f_q^{\text{decompress-compressed-pages-i}} \\ &= N_q^{\text{pages}} \cdot f_q^{\text{compressed-pages-i}} \cdot f_q^{\text{decompress-compressed-pages-i}} \end{aligned}$$

This allows one to factor out N_q^{pages} from the equation describing λ_q^{H2D},

$$\lambda_q^{\text{H2D}} = \frac{\sum_i f_q^{\text{compressed-pages-i}} \cdot f_q^{\text{decompress-compressed-pages-i}} \cdot S_{q,i}^{\text{page-compress}}}{\sum_i f_q^{\text{compressed-pages-i}} \cdot f_q^{\text{decompress-compressed-pages-i}} \cdot T^{\text{size}}_{\text{H2D}}\left(S_{q,i}^{\text{page-compress}}\right)}$$

Given this equation, one can deduce that λ_q^{H2D} is constant across different dataset sizes of the same type, whenever two conditions hold in this context. First of all, the distribution of page sizes within this column, must remain constant. Additionally, all ratios of page decompressions with respect to the amount of pages of each compressed page size within this column, must also remain constant. When there is no duplicate or skipping of page decompressions, each $f_q^{\text{decompress-compressed-pages-}i} = 1$ which means that we only require the first condition for constancy of λ_q^{H2D}.

5.1.2 Modelling the device to host transfer time

In a similar way we can write T_{D2H}, the time to transfer uncompressed pages back to the CPU, as

$$T_{\text{D2H}} = \sum_{q \in \mathcal{F}} \frac{S_q^{\text{uncompress}} \cdot f_q}{\lambda_q^{\text{D2H}}} \tag{5.8}$$

with

$$\lambda_q^{\text{D2H}} = \frac{\sum_i N_q^{\text{produced-decompressed-pages-i}} \cdot S_{q,i}^{\text{page-decompress}}}{\sum_i N_q^{\text{produced-decompressed-pages-i}} \cdot T_{\text{D2H}}^{\text{size}}\left(S_{q,i}^{\text{page-decompress}}\right)} \tag{5.9}$$

$$= \frac{\sum_i f_q^{\text{decompressed-pages-i}} \cdot f_q^{\text{produced-decompressed-pages-i}} \cdot S_{q,i}^{\text{page-decompress}}}{\sum_i f_q^{\text{decompressed-pages-i}} \cdot f_q^{\text{produced-decompressed-pages-i}} \cdot T_{\text{D2H}}^{\text{size}}\left(S_{q,i}^{\text{page-decompress}}\right)} \tag{5.10}$$

To understand the equations shown above, let us define $N_q^{\text{produced-decompressed-pages-i}}$ as the amount of produced uncompressed pages within column q of a size indexed by i. Whereas this amount might capture duplicate or the skipping of page decompressions, the parameter $N_q^{\text{uncompressed-pages-}i}$ simply counts the amount of pages of a size indexed by i, within column q. Let us also define N_q^{pages} as the amount of pages in column q. Additionally, note that $S_{q,i}^{\text{page-decompress}}$ refers to the decompressed page size in column q indexed by i. Using these expressions we can define all f parameters in equation 5.10 as follows.

$$f_q^{\text{produced-decompressed-pages-}i} = N_q^{\text{produced-decompressed-pages-}i} / N_q^{\text{uncompressed-pages-i}}$$

$$f_q^{\text{decompressed-pages-}i} = N_q^{\text{uncompressed-pages-}i} / N_q^{\text{pages}}$$

Assuming that the distribution of uncompressed page sizes is constant within a column for different data-set sizes of the same type, as well as that each $f_q^{\text{produced-decompressed-pages-i}}$ is constant across different data-set sizes of the same type, each λ_q^{D2H} is a constant calibratable parameter for a specific dataset type. As before, one does not need to satisfy the second condition, without duplicate or the skipping of page decompressions.

5.1.3 Rewriting T_{H2D}, T_{D2H}

Whereas the expressions in equation 5.8 and 5.4 define T_{D2H} and T_{H2D} we can rewrite them in yet another manner. We will start for T_{H2D}, by remembering that we replaced the denominator of 5.5 in 5.6, and also explained why this is true. Given equation 5.2, we can then formulate the following expression for T_{H2D}.

$$T_{\text{H2D}} = \sum_{q \in \mathcal{F}} \sum_{i \in c_q} N_q^{\text{decompress-compressed-pages-i}} \cdot T_{\text{H2D}}^{\text{size}}\left(S_{q,i}^{\text{page-compress}}\right) \tag{5.11}$$

Essentially, this equation computes the host to device transfer time, by outwardly summing over the transfer time per column q used in an RDataFrame program. Inwardly, we determine the transfer time for a column q by summing over the transfer time for each compressed page size occurring in it. We index these compressed page sizes by the parameter i. All compressed page size indexes for a column q reside in the set c_q. In previous equations, we did not explicitly formulate this set, because we always worked in the context of one specific column q.

A similar approach yields the following equation for T_{D2H}.

$$T_{\text{D2H}} = \sum_{q \in \mathcal{F}} \sum_{i \in u_q} N_q^{\text{produced-decompressed-pages-i}} \cdot T_{\text{D2H}}^{\text{size}} \left(S_{q,i}^{\text{page-decompress}} \right) \tag{5.12}$$

Note that u_q now refers to the set of indexes corresponding to the uncompressed page sizes occurring in column q.

5.1.4 Defining S_{H2D}, S_{D2H}, λ^{D2H} and λ^{H2D}

Intuitively, stripping the transfer time functions from their argument in equations 5.11 and 5.12 will yield the total amount of data transferred from the host to the device and vice versa.

$$S_{\text{H2D}} = \sum_{q \in \mathcal{F}} \sum_{i \in c_q} N_q^{\text{decompress-compressed-pages-i}} \cdot S_{q,i}^{\text{page-compress}} \tag{5.13}$$

$$S_{\text{D2H}} = \sum_{q \in \mathcal{F}} \sum_{i \in u_q} N_q^{\text{produced-decompressed-pages-i}} \cdot S_{q,i}^{\text{page-decompress}} \tag{5.14}$$

Using these equations, as well as 5.11 and 5.12 we can formulate the average data transfer speed λ^{H2D} from the host to device and from the device to the host λ^{D2H}.

$$\lambda^{\text{H2D}} = S_{\text{H2D}} / T_{\text{H2D}} \tag{5.15}$$

$$\lambda^{\text{D2H}} = S_{\text{D2H}} / T_{\text{D2H}} \tag{5.16}$$

We will now try to relate these average data transfer speeds to the data transfer speeds per column, starting with λ^{H2D}. One can rewrite equation 5.15 as $T_{\text{H2D}} = S_{\text{H2D}} / \lambda^{\text{H2D}}$. This new expression and equation 5.3 serve as a starting point for deriving the desired relation.

$$\frac{S_{\text{H2D}}}{\lambda^{\text{H2D}}} = \sum_{q \in \mathcal{F}} \frac{S_q^{\text{compress}} f_q}{\lambda_q^{\text{H2D}}}$$

$$\lambda^{\text{H2D}} = \frac{S_{\text{H2D}}}{\sum_{q \in \mathcal{F}} \frac{S_q^{\text{compress}} f_q}{\lambda_q^{\text{H2D}}}}$$

$$\lambda^{\text{H2D}} = \frac{1}{\sum_{q \in \mathcal{F}} \frac{S_q^{\text{compress}} f_q}{S_{\text{H2D}}} \frac{1}{\lambda_q^{\text{H2D}}}}$$

Apart from telling us how to compute the average host to device transfer speed from the corresponding column wise speeds, the last equation also allows us to state the conditions under which it remains unchanged. More specifically, λ^{H2D} remains constant for a dataset of a certain type, regardless of its size, whenever each fraction $\frac{S_q^{\text{compress}} f_q}{S_{\text{H2D}}}$ and λ_q^{H2D} stays constant in under these conditions. For a column-wise transfer speed λ_q^{H2D}, remember that we previously specified when it is constant for a certain dataset type.

Each fraction refers to the ratio of transferred compressed data for a certain column, with respect to the total amount of transferred compressed data. Without duplicate or the skipping of page decompressions, we transfer each compressed page belonging to used columns, exactly once. In this case, S_{H2D} thus refers to the combined size of all compressed pages, belonging to used columns. Additionally, each f_q is then one, so the fraction characterises the compressed size of a column, with respect to the combined compressed size of all used columns.

For λ^{D2H} we can deduce the following equation, using a similar approach.

$$\lambda^{\text{D2H}} = \frac{1}{\sum_{q \in \cup_{f=1}^{F} F_f} \frac{S_q^{\text{uncompress}} \cdot f_q}{S^{\text{D2H}}} \frac{1}{\lambda_q^{\text{D2H}}}}$$

Yet again, λ^{D2H} remains constant, whenever each fraction and λ_q^{D2H} do. In general, such a fraction now refers to the amount of decompressed data for a column q, transferred back to the CPU, with respect to the total amount of decompressed data transferred back to the CPU. Without skipped or duplicate page decompressions, each fraction refers to the uncompressed size of a column with respect to the combined uncompressed size of all columns belonging to used fields.

5.1.5 Transfer time/speed predictions

In order to evaluate the performance of transferring individual pages, we will first determine the transfer speeds and transfer times, for each dataset that we used in the single-threaded CPU decompression model calibration/verification. For the same goal, we will also determine the transfer times for each dataset used in the multi-threaded CPU decompression model verification. We will then compare the predicted transfer times to CPU decompression latencies, to evaluate whether the performance of transferring individual pages, is sufficient. More specifically, we will compare the predicted transfer time for each dataset used in the single-threaded calibration/verification, to the corresponding single-threaded decompression times. Similarly, we will compare the predicted transfer time

for each dataset used in multi-threaded verification, to the corresponding multi-threaded decompression times, across various thread counts.

5.1.5.1 Determining decompressed page distributions

For each considered dataset, we predict transfer speeds and times using equations 5.15, 5.16, 5.14, 5.13, 5.12, and 5.11. When evaluating equations 5.12 and 5.14, one can notice that we need a distribution of decompressed page sizes per used column. Additionally, equations 5.11, and 5.13 show that, per used column, we need a distribution of decompressed pages by their compressed size. Given that the latter distribution depends on the compression algorithm efficiency, we expect the host to device transfer speed and time to differ across various compression algorithm/level settings. For the device to host transfer speed and time we do not have a clear expectation.

Remember that our current offloading idea replaces the routine of decompressing a page in the single-threaded implementation, with moving it to the GPU, decompressing it there, and finally moving it back. We can thus obtain the two distributions for each used column of a considered dataset, by keeping track of the uncompressed and compressed page sizes in the process of decompression within the current single-threaded implementation.

In order to do this, we added instrumentation within the `RPageSourceFile` class of the ROOT framework (81). Within such a page source, we store the two required distributions for each used column in two dictionaries, where the compressed/decompressed page size serves as a key, and the frequency as value. We build all distributions incrementally whenever the framework decompresses a page within the `PopulatePageFromCluster` method of the `RPageSourceFile` instantiation (82). With this we mean that whenever the framework decompresses such a page, we increment the occurrence of its uncompressed and compressed size, in the two dictionaries of its encompassing column.

We already added this instrumentation when we performed the single and multi-threaded CPU decompression model calibration/verification. In order for us to obtain the two distributions per used column, for each dataset used in the single-threaded calibration/verification, we could simply read the distributions from each measurement taken there. For the multi-threaded verification, we measured the multi-threaded decompression time for each dataset, across various thread counts. These measurements also contain two distributions for each used column, separately for each thread. Taking the measurement of a dataset, with only one thread, gives us the desired two distributions per used column, for the single-threaded

implementation. The instrumentation produces separate distributions across threads, because the RDataFrame framework creates a separate `RPageSourceFile` instantiation per thread.

5.1.5.2 Benchmarking the H2D and D2H transfers

Now that we know the distributions, for each dataset used in the single-threaded calibration/verification, and the multi-threaded verification, we only need to determine the functions $T^{\text{size}}_{\text{D2H}}$ and $T^{\text{size}}_{\text{H2D}}$. We need to know those functions to calculate each T_{H2D} and T_{D2H} for each considered dataset. Additionally, we need those functions to calculate λ_{H2D}, λ_{D2H} for each dataset used in the single-threaded calibration/verification.

More specifically, we need to know the values of these functions, for each possible transferred page size, across all considered datasets. As such, we first determined the range of possible page sizes for each function. In other words, we calculated the minimum and maximum for both the uncompressed and compressed page size distributions, across all considered datasets. Subsequently, we benchmarked the host to device transfer time for a number of a samples between the minimum and maximum compressed page size. We also performed this procedure, for the device to host transfer time. The code listing shown below displays an important part of the host to device transfer time benchmark.

```
for (int run = 0; run < 2; run++) {
    unsigned long cursize = minsize;
    for (int i = 0; i < steps; i++) {
        cudaEventRecord(before);
        for (int j = 0; j < repetitions; j++)
            cudaMemcpy(gpumemory, cpumemory, cursize, cudaMemcpyHostToDevice);
        cudaEventRecord(after);
        cudaEventSynchronize(after);
        cudaEventElapsedTime(&latencies[i], before, after);
        sizes[i] = cursize;
        cursize = (unsigned long ) ((double) minsize + stepsize * ((double) i + 1.0f));
```

As one can see in this listing, this benchmark measures the average `cudaMemcpy` latency for host to device transfers, for a number of steps between a minimum and maximum transfer size that one can specify as a command line argument. The outer loop reveals that we perform these measurements twice, and only use the results of the second round. We don't use the measurements of the first round, because initial experimentation revealed that a number of the first data transfers in this round, do not yet reach their optimal speeds. For benchmarking the device to host transfer times, we use code that is equivalent, apart from the data transfer direction parameter within the `cudaMemcpy` call.

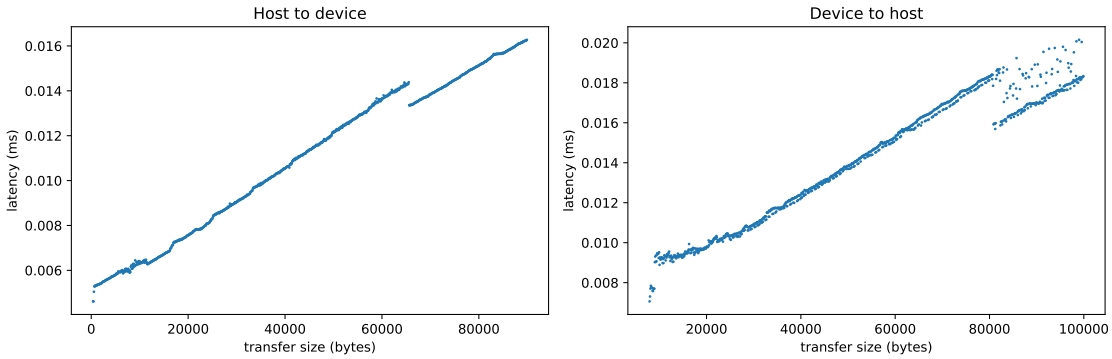


Figure 5.1: Displays average (n=1000) D2H and H2D transfer times from 2933 MT/sec main memory to an NVIDIA A6000 over PCIE 4.0x16, on a DAS6 node.

We present the found average host to device and device to host transfer times over an PCIE 4.0x16 link, for an NVIDIA A6000 in figure 5.1. For these measurements we used a DAS6 node, for which we specify all other relevant hardware in table 7.1. We determined these average transfer times (n=1000) in both directions for 1000 steps between the minimum and maximum values that each x-axis indicates. Using these samples we can define $T^{\text{size}}_{\text{D2H}}$ and $T^{\text{size}}_{\text{H2D}}$ as follows. Whenever we pass a size for which we measured an average latency, we can return this latency. Otherwise, this size lies between two sizes for which we know the average latency. As such we then linearly interpolate between those latencies.

5.1.5.3 Comparing single-threaded CPU decompression time with the predicted transfer time

Using these newly defined functions $T^{\text{size}}_{\text{D2H}}$ and $T^{\text{size}}_{\text{H2D}}$, as well as the relevant distributions for each dataset used in the single threaded calibration/verification, we calculated each corresponding λ_{H2D}, λ_{D2H}, T_{H2D} and T_{D2H}. Figure 5.2 displays the calculated sums of T_{H2D} and T_{D2H} separately for each considered compression algorithm/level, together with each calibrated single-threaded decompression time model. We display the computed sums as points, where the dotted lines linearly interpolate between these points, and we show the calibrated models as solid lines. It seems that fitted linear models $ax+b$ adequately capture the predicted transfer times for each compression algorithm, because they overestimate and underestimate the predictions by at most 0.16% and -0.44%, respectively.

Whereas we calibrated each decompression time model on a node provided by CERN, we determined $T^{\text{size}}_{\text{D2H}}$ and $T^{\text{size}}_{\text{H2D}}$ on a DAS6 node. See table 7.1 for hardware details regarding both nodes. We could not determine both size functions on the CERN node, because

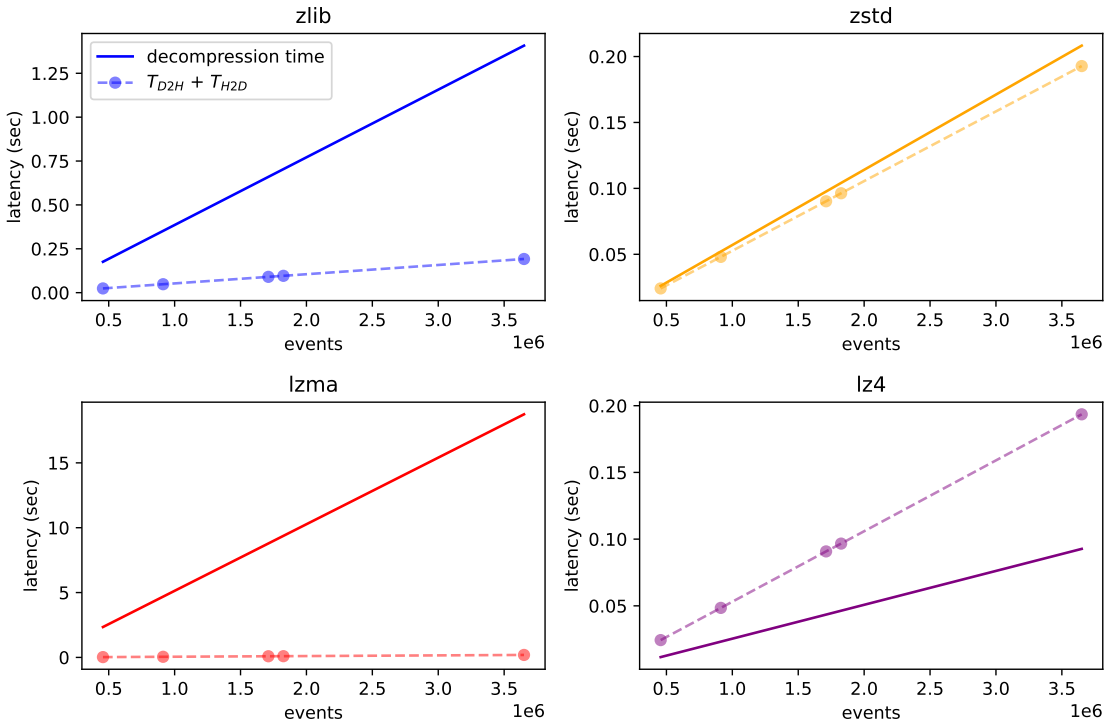


Figure 5.2: For each compression algorithm and level combination accepted by the LHCB benchmark, we show the calibrated single-threaded model as a solid line. In each graph, the points indicate corresponding predicted transfer times ($T_{\text{H2D}} + T_{\text{D2H}}$), with the dotted lines linearly interpolating these points.

even though it contains an NVIDIA GPU, installing CUDA would cause other software to break. When comparing each calibrated single-threaded model in figure 5.2 to the predicted transfer times, we thus assume that both $T^{\text{size}}_{\text{D2H}}$ and $T^{\text{size}}_{\text{H2D}}$ would equal those on the CERN node, if we could have used an NVIDIA A6000 GPU, linked with PCIE4.0x16. A factor potentially weakening this assumption, is the 3200 MT/sec DAS6 node main memory not matching the 2933 MT/sec CERN node main memory speed.

Comparing the predicted transfer times with the single-threaded CPU decompression times, for each compression algorithm, results in a mixed view. For ZLIB and LZMA, transferring individual pages takes a small amount of time with respect to CPU decompression. On the other hand, for ZSTD, we observe a very narrow margin between the CPU decompression time and the predicted transfer times. Finally, for LZ4, transfer time dominates the CPU decompression time. In the latter case, GPU decompression on the basis of individual page transfer, will not improve the single-threaded LHCB performance. The separate transfer time graphs per transfer direction, for each dataset used in the single-

threaded calibration/verification, shown in figure 5.4, demonstrate that the varying gaps across graphs in figure 5.2 actually result from differences in the decompression times across the algorithms. We can conclude this because the transfer time lines in either direction, do not vary that much, as a result of compression algorithm choices.

Visually, the gap between $T_{\text{D2H}} + T_{\text{H2D}}$ and the CPU decompression time in figure 5.2, seems to increase with increasing dataset sizes for the ZLIB, ZSTD and LZMA algorithms. This might hint on the individual page transfer time being less dominant for bigger datasets, implying we could get a better performance of this approach, for bigger datasets. Calculating the fraction between both quantities tells a different story, however.

Because we can model the transfer time and decompression time, as linear functions $ax+b$, the percentage of the single-threaded CPU decompression time spent on transferring data, converges towards a single asymptote. See the section 7.3 in the Appendix, for a proof of this claim. For ZLIB, ZSTD, LZMA, and LZ4 these asymptote percentages equal a fixed 13%, 93% 1%, and 208%, respectively. When calculating these percentages for the datasets of each compression algorithm/level used in the single-threaded calibration/verification, we get values that deviate between -0.32% and 0.80% from these asymptotic percentages.

A bigger dataset size will thus not decrease the percentage of the single-threaded CPU decompression time that our current GPU offloading decompression approach needs to spend on transferring data. As such we can conclude that for the LHCB benchmark and datasets of the B2HHH type, offloading decompression to the GPU on the basis of transferring individual pages, will potentially only outperform single-threaded CPU decompression for ZLIB and LZMA.

5.1.5.4 Comparing multi-threaded CPU decompression time with the predicted transfer time

We will now compare the transfer times of this method with the decompression latency within the multi-threaded CPU decompression implementation. For this, let us first recall the multi-threaded verification process. In this process, we measured the multi-threaded LHCB benchmark latency spent on decompression in the slowest thread, on a segment of the B2HHH dataset, compressed with ZLIB, ZSTD, LZMA or LZ4, across various thread counts. We then compared the predictions of all calibrated multi-threaded models, with these measured values.

For each dataset used in this verification, we again predict its transfer time according to equations 5.12 and 5.11 using the fitted transfer time functions, as well as the obtained decompressed page distributions. For each compression algorithm, for which there is a

separate dataset, we then compute the fraction of the predicted transfer time, with respect to the multi-threaded decompression latency across a thread count range. We present these fractions, in figure 5.3. It is important to note that in all subsequent analysis of these fractions, we again assume that the fitted transfer time functions on the DAS6 node, equal those on the CERN node, if we could have used an NVIDIA A6000 there.

Even though not visible in the graphs of this figure, the fractions for one thread, roughly correspond to the asymptotic fractions that we determined earlier for each compression algorithm, when comparing the single-threaded decompression times with the predicted transfer times. More specifically, we get values of 14%, 97%, 1% and 216% for ZLIB, ZSTD, LZMA and LZ4, respectively. These numbers confirm our previous conclusion, that GPU offloading with transferring individual pages, does not outperform single-threaded decompression for the LHCB benchmark, when using ZSTD and LZ4.

Increasing the thread count only decreases the CPU decompression time, while not impacting the transfer time, so obviously, increasing thread counts only increase percentage of CPU decompression time spent on transfers. As such, GPU decompression offloading on the basis of transferring individual pages, is also not beneficial with respect to multi-threaded decompression within the LHCB benchmark, for ZSTD and LZ4.

Whereas the fraction of the CPU decompression time is small for one thread, in the case of ZLIB, figure 5.3 shows that it exceeds one for 9 threads. Even though GPU decompression performance, also determines the performance of GPU decompression offloading with respect to CPU decompression, GPU offloading can potentially outperform multi-threaded CPU decompression with ZLIB, for a number of threads lower than 9. We expect this amount of threads to be around the low-end of this range. For LZMA, figure 5.3 shows that even for the highest amount of threads for which we took measurements, individual page transfers only take up 20% of the time spent of CPU decompression. Of all tested compression algorithms, LZMA thus likely has the most promising performance improvement of GPU decompression offloading on the basis of transferring individual pages, with respect to multi-threaded decompression.

5.1.5.5 The variation in transfer speeds/times across compression algorithms

Remember that we expected the H2D transfer times and transfer speeds to vary across compression algorithms, because the distributions of decompressed pages by their compressed size, depends on compression algorithm efficiency. The H2D transfer time function in equation 5.11 can not drive this variation, because it only depends on hardware involved in its calibration, which are the NVIDIA A6000, PCIE4.0x16 and 2933 MT/sec DDR4 RAM. In

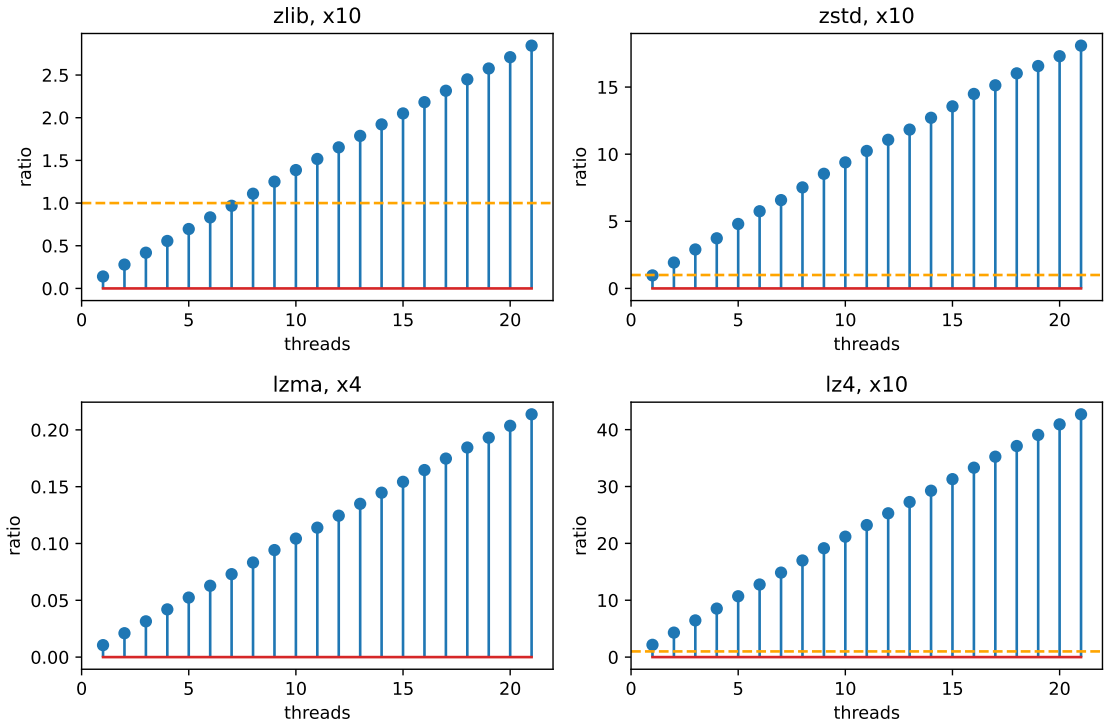


Figure 5.3: Each graph shows the fraction of the predicted time spent on transferring individual pages with respect to the multi-threaded CPU decompression latency for the LHCB benchmark, on a dataset compressed with a certain compression algorithm, across various thread counts. All datasets used here, are a segment of events 0 to 684494 from the B2HHH dataset, compressed with a certain algorithm, duplicated 4 or 10 times. The title of each graph describes the used compression algorithm, as well as this duplication factor. The multi-threaded CPU decompression latencies originate from the multi-threaded model verification in chapter 5, whereas we predict the individual page transfer times here, for an NVIDIA A6000 linked with main memory using PCIE4.0x16, on a DAS6 node.

order to test our hypothesis, we use figure 5.4, which shows transfer speeds and transfer times, for all datasets originating from the single-threaded calibration/verification.

When evaluating the H2D transfer time graph in figure 5.4 we can indeed see the LZMA, ZSTD, and LZ4 seperately. More precisely, the LZMA line has the lowest slope, followed by the ZSTD and the LZ4 line. This ordering corresponds with the observed ordering in the H2D transfer speed graph in figure 5.4. Because the ZLIB line lies very close to the ZSTD line, they overlap in the H2D transfer time graph.

As a proof of the link between compression algorithm efficiency and the variation between both the H2D transfer times and H2D transfer speeds, we can evaluate the compression ratios of each algorithm, in their relation to the ordering of the H2D transfer speeds and

transfer times. Across all considered datasets sets, the LZMA, ZSTD, ZLIB and LZ4 algorithm realise an average (n=5) compression ratio of 1.53 (var=5.80e-7), 1.46 (var=1.10e-7), 1.45 (var=4.14e-7), and 1.41 (var=2.83e-7), for all used columns, respectively. Compression ratio's thus seem inversely proportional to the realised H2D transfer speeds and transfer times. This makes sense, as a lower compression ratio realises bigger pages, for which there is a higher transfer speed than for smaller pages.

Visually, it seems that the D2H transfer time lines mostly overlap. Evaluating the D2H transfer speeds reveals small albeit distinguishable difference between the algorithms, that grow with the dataset size. Given the constancy of $T^{\text{size}}_{\text{D2H}}$ function across all algorithms of this experiment, a difference of the decompressed page size distributions by their uncompressed sizes, drives this difference. The exact mechanism by which the choice of an compression algorithm influences these distributions, is unclear.

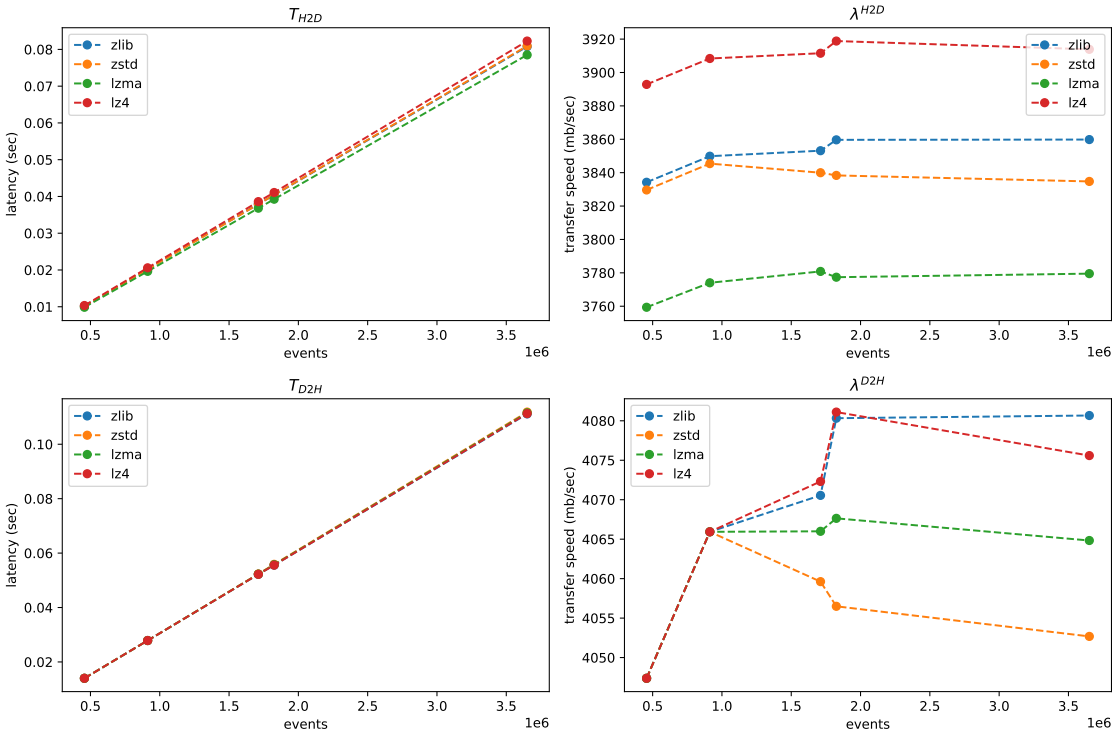


Figure 5.4: The left and right column of graphs display the predicted H2D and D2H transfer times, and the predicted H2D and D2H transfer speeds, respectively, for all datasets used in the single threaded LHCB benchmark calibration/verification, categorized by their compression algorithm/level, on a DAS6 node. Each point signifies a predicted value for a dataset with a certain amount of events. Dotted lines linearly interpolate predicted values for a specific compression algorithm.

5.1.5.6 Cause of low transfer speeds and next steps

Up till this point in time, we have seen that offloading decompression to the GPU on the basis of individual page transfers, can potentially improve the performance of the single-threaded LHCB benchmark, for LZMA and ZLIB. For these algorithms, we also saw that our GPU decompression offloading approach, can potentially outperform the LHCB benchmark with multiple threads. For ZLIB this is likely an amount of threads, in the low-end of the range up till and including 9 threads. On the other hand, LZMA seems promising for all considered thread counts, which are all thread counts up till and including 21 threads. We use careful language, because all claims are estimates on the basis of the fraction of CPU decompression time, that the GPU offloading implementation needs to spend on data transfers, not considering the time that the GPU needs to spend on decompression. Also note that we made all these observations, for a hypothetical node with an AMD EPYC 7702P processor running at 2.0GHZ, with DDR4 main memory, connected with an NVIDIA A6000 using PCIE4.0x16.

When looking at the realised transfer speeds in figure 5.4, for all single-threaded calibration/verification sets, we can see that they reach approximately 4 GB/sec for H2D and D2H transfers. Transfers in either direction thus under-utilise the potentially 32 GB/sec transfer bandwidth offered by PCIE4.0x16(83).

The main reason for this is the small granularity with which we transfer data. As evidence for this claim, we can see in figure 5.4, that the transfer speeds for uncompressed pages are bigger than that of compressed pages. Additionally, we saw that the lowest compression ratios with the biggest pages yielded the highest H2D transfer speeds, for all single-threaded calibration/verification sets.

We will thus explore a new data transfer approach that increases the data transfer granularity, so that we better utilise the available bandwidth. Lower transfer times will allow a greater potential performance benefit of GPU offloading, with respect to single or multi-threaded decompression, for the LHCB benchmark. More specifically, for LZ4 and ZSTD, a potential performance benefit might be possible for one or more threads. Additionally, a potential performance improvement might extend to more threads than currently possible, for ZLIB, and nevertheless improve across thread counts in general. Finally, for LZMA, the potential performance improvement will improve over all thread counts.

5.2 Buffered page transfers

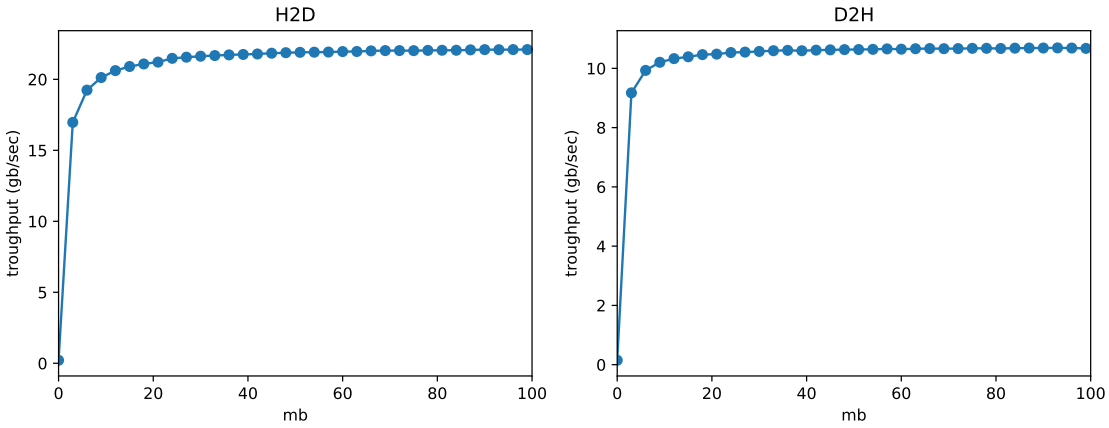


Figure 5.5: D2H and H2D transfer speeds for an NVIDIA A6000 GPU connected with 128 GB DDR4 RAM (2933 MT/sec) using PCIE4.0x16, on a DAS6 node.

Using the same benchmark that we used in section 5.1.5.2, we measured the D2H and H2D transfer times for bigger transfer sizes, yet again on a DAS6 node. With this data we calculate the corresponding H2D and D2H transfer speeds. In figure 5.5 we can see the calculated D2H and H2D speeds seemingly converging to an asymptote. Because both sequences are not strictly increasing, we take the highest value of both as their maximum transfer speed. For the H2D transfer speed this is 22.32 GB/sec and for the D2H transfer speed this is 10.74 GB/sec. Transfer speeds reach at least 95% of these values, for data segments bigger than 21 MB and 9 MB, for H2D and D2H transfers, respectively.

Whereas the page transfer approach described in the previous section, transferred each page to the GPU separately, we can also write many compressed pages in a fixed-size buffer on the CPU, before transferring it to the GPU. After decompressing each of the pages in such a buffer, we could then transfer all decompressed pages back to the CPU, at once. We can set the size of the compressed page buffer, so that we reach most of the D2H and H2D bandwidth. In this way, we also reach a high D2H bandwidth, with a worst-case compression ratio of one.

For our hardware, we should set the compressed buffer size to 21 MB, so that we utilise at least 95% of the 22.32 GB/sec H2D and the 10.74 GB/sec D2H bandwidth. Compared to the individual page transfers, this improves the H2D transfer speeds by 5.4x and the D2H transfer speeds with 2.6x. [1]

[1] Here we use 4 GB/sec as a reference for the D2H and H2D individual page transfer speed.

5.2.1 Transfer time model

We can also predict the transfer time of this approach, for both H2D and D2H transfers, using the following equations.

$$T_{\text{H2D}} = \frac{\lfloor S^{\text{compressed}}_{\text{transferred}}/C \rfloor \cdot C}{\lambda^{\text{H2D}}(C)} + \frac{S^{\text{compressed}}_{\text{transferred}} \% C}{\lambda^{\text{H2D}}(S^{\text{compressed}}_{\text{transferred}} \% C)} \tag{5.17}$$

$$T_{\text{D2H}} = \frac{\lfloor S^{\text{compressed}}_{\text{transferred}}/C \rfloor \cdot C \cdot r}{\lambda^{\text{D2H}}(C \cdot r)} + \frac{S^{\text{uncompressed}}_{\text{transferred}} - \lfloor S^{\text{compressed}}_{\text{transferred}}/C \rfloor \cdot C \cdot r}{\lambda^{\text{D2H}}(S^{\text{uncompressed}}_{\text{transferred}} - \lfloor S^{\text{compressed}}_{\text{transferred}}/C \rfloor \cdot C \cdot r)} \tag{5.18}$$

Given a fixed-size compressed buffer of size C, the first term in T_{H2D} models the transfer time of all full buffers, whereas the second term models the transfer time for remaining data. This simplifies the reality wherein we prevalently do not fill such buffers fully. This occurs whenever the buffer does not hold enough space to hold the next compressed page that needs decompression. Both terms in this equation, calculate a transfer time contribution by dividing a size, by a transfer speed. All transfer speeds result from a corresponding λ function, that takes a transfer size as an argument.

In the equation describing T_{D2H}, the first term models the transfer time for the decompressed counterparts of each "full" compressed buffer. On the other hand, the second term models the transfer time of the decompressed counterpart of the remaining compressed data, that did not fit in full buffers. To calculate the decompressed size of all full buffers of compressed pages, we use the compression ratio r.

We can use these equations, to calculate the predicted transfer time on our hardware, with compressed buffers of 21MB, yet again, for each single-threaded LHCB calibration/verification dataset as well as each multi-threaded LHCB verification set. With these results, and those of the previous section, we can compare this approach with transferring each compressed page individually.

5.2.1.1 Modelling λ^{H2D} and λ^{D2H}

In order to do so, we will first of all determine the functions λ^{H2D} and λ^{D2H} for our hardware, by fitting functions to the data, partly shown in figure 5.5. We already knew that functions of the form $f(x) = c - ae^{-b\sqrt{x}}$, look like the data in figure 5.5, for $a, b, c > 0$. We found this parameter range by creating plots for various values of a, b and c. Figure 5.6 shows a plot for $a, b, c = 1$, which indeed resembles the data in figure 5.5. It starts at $x = 0$, and then increases towards an horizontal asymptote.

This observation turns out to be true for $f(x)$ in general, whenever $a, b, c > 0$. It is easy to understand that $f(x)$, does not have values for $x < 0$, given the presence of a square

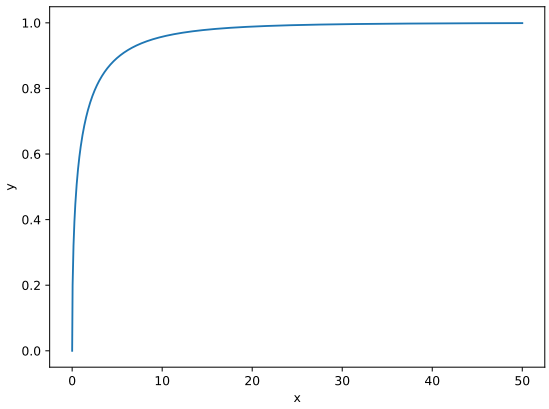


Figure 5.6: Plot of the function $y = 1 - e^{-\sqrt{x}}$.

root in this function. By evaluating $\lim_{x\to\infty}\left[c - ae^{-b\sqrt{x}}\right]$ one can determine the presence of a horizontal asymptote, with value c. We can also proof that $f(x)$ strictly increases from $x \geq 0$ towards this asymptote, which we do in section 7.4 of the appendix.

For our purposes, wherein transfer bandwidths can never be negative, it is important to ensure that $f(x) \geq 0$. Because we know $f(x)$ is strictly increasing in the context of $a, b, c > 0$ and $x \geq 0$, this is equivalent to ensuring that $f(0) \geq 0$. This statement is true if and only if $a \leq c$ is true. Thus $f(x) \geq 0$ if $a \leq c$. By rewriting a as βc we can re-express this statement as $f(x) \geq 0$ if $0 < \beta \leq 1$. One can deduce the tail of this double inequality as follows $a \leq c \leftrightarrow \beta c \leq c \leftrightarrow \beta \leq 1$. Its head is necessary to ensure that $c > 0$.

5.2.1.2 Fitting $\lambda^{\mathbf{H2D}}$ and $\lambda^{\mathbf{D2H}}$

We fit a model $f(x)$ to the H2D and D2H transfer bandwidth data, party displayed in figure 5.5, using the Python `curve_fit` method, provided by `scipy`. This method allows us to specify our function of the form $y = c - \beta c e^{-b\sqrt{x}}$, the y and x data to which we fit our function, as well as bounds on the parameters c, b and β. It uses a least squares method to determine the parameters. As explained previously, we limit β as $0 < \beta \leq 1$. Given that c determines the transfer speed to which the function eventually converges, we bound it between 20 GB/sec and 30 GB/sec for H2D transfers, and 10 GB/sec and 20 GB/sec for D2H transfers, which is intuitive when we look at figure 5.5. We simply bound b as having a value greater than 0. Earlier we determined this condition to be necessary, just for $f(x)$ to look like the transfer bandwidth data. More specifically, this condition is necessary for $f(x)$ to consist of a horizontal asymptote. Note that because `curve_fit` only offers inclusive inequalities to specify parameter bounds, we use a small delta, to specify all

exclusive inequalities. For example, we specify the lower bound of b as $0 + \delta$ where we set δ as $1 \cdot 10^{-100}$. We present the obtained parameters for the H2D and D2H transfer speeds

	c	b	β
H2D	22.29	0.7646	1
D2H	10.73	1.042	1

Table 5.1: Fitted function parameters of $f(x) = c - \beta c e^{-b\sqrt{x}}$ to the H2D as well as D2H transfer speeds, partly shown in figure 5.5.

models, in table 5.1. Both models have a high deviation for the smallest transfer size for which we took measurements. Remember that we define a deviation as $(M/R - 1) \cdot 100\%$ with M a model prediction and R a reference value. For the smallest size of 0.001 MB we have a deviation of 151.15% and -57.24%, for the D2H and H2D transfer speed model, respectively. For all other transfer sizes, the deviation is always between -3.58% and 2.29% when considering both models in unison. Because of this, we still choose to use these calibrated models to predict transfer times when using a buffer of compressed pages.

5.2.1.3 Obtaining $S^{\text{uncompressed}}_{\text{transferred}}$, $S^{\text{compressed}}_{\text{transferred}}$ and r for all considered datasets

In order for us to calculate the predicted transfer times with equations 5.17 and 5.18, for each single-threaded calibration/verification LHCB dataset as well as each multi-threaded LHCB verification dataset, we still need to obtain some data. More specifically, we need to obtain the amount of transferred uncompressed data $S^{\text{uncompressed}}_{\text{transferred}}$ and the amount of transferred compressed data $S^{\text{compressed}}_{\text{transferred}}$ for all used columns in each considered dataset. Given a dataset and these numbers, we can also compute the required compression ratio r of all used columns with $r = S^{\text{uncompressed}}_{\text{transferred}} / S^{\text{compressed}}_{\text{transferred}}$.

Remember that we have measurements from the single-threaded calibration/verification that describe the distribution of all decompressed pages belonging to used columns, by their uncompressed and compressed size, for each single-threaded calibration/verification dataset. Simply summing the page distributions by their compressed and uncompressed size, yields the required values of $S^{\text{uncompressed}}_{\text{transferred}}$, $S^{\text{compressed}}_{\text{transferred}}$ and r, for each single-threaded calibration/verification dataset.

For the multi-threaded verification datasets, we can obtain these values in a similar manner. Recall that within the multi-threaded verification, we used one separate dataset for each compression algorithm, and measured the decompression latency for each algorithm, across various thread counts. These measurements also contain decompressed page size

distributions, for each column, separately for each thread. We can use the distributions of each multi-threaded verification dataset, from the corresponding measurements with one thread, to calculate each $S^{\text{uncompressed}}_{\text{transferred}}$, $S^{\text{compressed}}_{\text{transferred}}$ and r. We take the decompressed page distributions from measurements with one thread, because we assume our GPU offloading implementation to be a modification of the single-threaded CPU decompression implementation. More specifically, due to this assumption, we know that the distributions of decompressed pages, by their uncompressed and compressed size, for each used column, equal the distributions of transferred uncompressed and compressed page sizes.

5.2.1.4 Performance of buffered data transfers in a single-threaded context

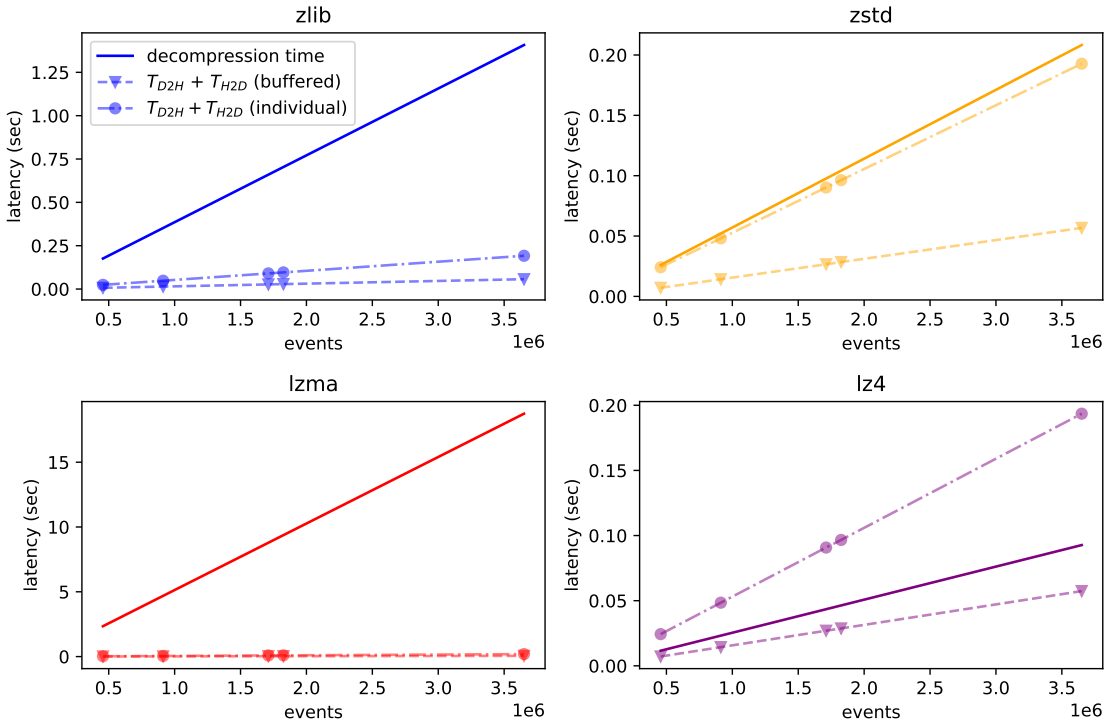


Figure 5.7: We show a graph for each compression algorithm/level combination accepted by the LHCB benchmark, wherein the solid line displays a corresponding calibrated single-threaded CPU decompression time model. Additionally, the circles display the predicted transfer times of each dataset corresponding to that algorithm/level, if we transfer pages individually. Finally, the triangles display the transfer times, if we use a compressed page buffer of 21 MB. We calibrated the single-threaded decompression time model on a node provided by CERN, whereas we calibrated the GPU transfer time models on a DAS6 node with an NVIDIA A6000. All dashed lines linearly interpolate between the circles or triangles.

Figure 5.7 shows the predicted transfer times when transferring compressed pages in a

buffer of 21 MB, as triangular points, for each compression algorithm separately. More specifically, for each compression algorithm accepted by the LHCB benchmark, we display these predicted transfer times for each corresponding dataset used in the single-threaded verification/calibration. We also display the calibrated single-threaded decompression time model for each compression algorithm, as a straight line, as well as the predicted transfer time when transferring individual pages, using circles and their interpolating dashed lines. In the figure, one can easily observe that buffered page transfers outperform individual page transfers for ZLIB, ZSTD and LZ4. For LZMA this is also the case, but not visible due to the relative magnitude of the single-threaded decompression times. Whereas previously, individual page transfers for ZSTD almost cost the same time as performing single-threaded decompression on the CPU, this time is now significantly lower. Moreover, buffered page transfers are now faster than single-threaded CPU decompression for the LZ4 compression algorithm.

The ratios of buffered page transfer times with respect to the corresponding single-threaded CPU decompression times, are constant across the single-threaded calibration/verification datasets for each considered compression algorithm. For ZLIB, ZSTD, LZMA and LZ4 those have the values of 4%, 27%, 0.30% and 62%, respectively. This significantly improves upon the ratios of 13%, 93%, 1%, and 208% achieved for individual page transfer times with respect to the single-threaded CPU decompression times.

5.2.1.5 Performance of buffered data transfers in a multi-threaded context

Now we will evaluate the performance of buffered data transfers to and from the GPU, for the LHCB benchmark, in a multi-threaded context. For this we first of all compute the ratios of the buffered transfer time, with respect to multi-threaded decompression times, for each multi-threaded verification dataset, across various thread counts.

Figure 5.8 shows these fractions as orange bars, whereas it shows the corresponding individual page transfer fractions in blue. Yet again, buffered data transfer fractions for one thread are equal to those found in the previous single-threaded evaluation. Because buffered data transfers reduce the transfer time, we can see that the corresponding fractions have lower values than those of individual page transfers, across all algorithms and thread counts. This is also true for thread counts not displayed for ZSTD and LZ4.

Unfortunately, this is not enough for GPU LZ4 decompression, on the basis of buffered page transfers, to outperform CPU LHCB decompression in a multi-threaded environment. That is because its fractions in figure 5.8 are then visibly bigger than one. We can see a buffered page transfer time fraction smaller than one, when using two threads for ZSTD

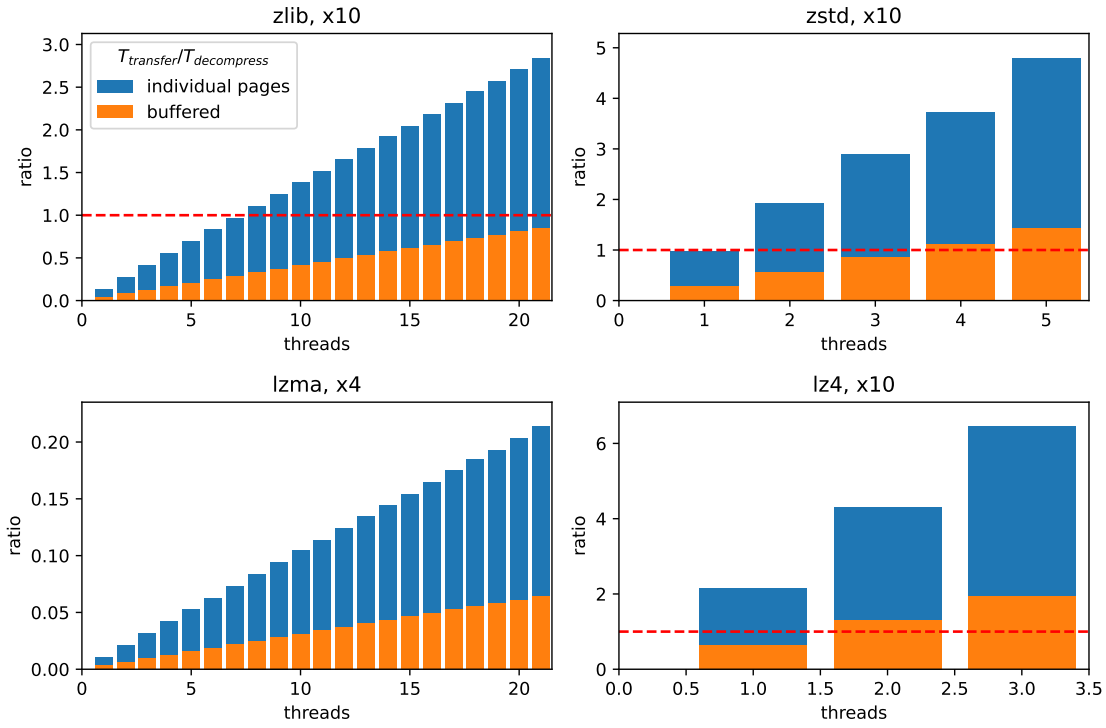


Figure 5.8: Each graph displays fractions of multi-threaded LHCB decompression times, spent on transferring data to and from the GPU, for a multi-threaded decompression verification dataset compressed with one of the four algorithms. More specifically, within each graph we plot the fractions of individual page transfer times and buffered transfer times with respect to multi-threaded decompression times, in blue and orange, respectively.

decompression. In contrast to individual page transfers, GPU offloading can thus potentially improve the performance of multi-threaded ZSTD decompression. In the same way, one can see that the range of thread counts for which GPU offloading can potentially improve the performance of multi-threaded ZLIB decompression, extends to all evaluated thread counts, with the usage of buffered page transfers. Finally, whereas a performance improvement is visibly possible for LZMA GPU decompression by transferring individual pages, it has become more likely or higher by transferring buffers of pages, due to the visibly reduced fractions across all evaluated thread counts.

5.3 GPU decompression speeds

Up till now we have defined T_{D2H} and T_{H2D} for GPU decompression offloading using buffered page transfers and individual page transfers. It turned out that buffered page transfers yielded a superior data transfer performance for the LHCB benchmark, in a multi-threaded and single-threaded context. As such, we will from now on study GPU offloading on the basis of this transfer method. We will see that transferring a buffer of compressed data is also advantageous due to its effect on the potential GPU decompression performance, as this relies on the ability to decompress many compressed data segments in parallel.

5.3.1 NVComp

We model the GPU decompression performance after `nvcomp` 3.01, a well-known and high-performing CUDA compression/decompression library by NVIDIA that is easy to setup and use (84, 85). Unfortunately, this library only allows us to decompress data compressed with ZSTD, ZLIB and LZ4, meaning that we cannot evaluate the hypothetical performance of GPU LZMA decompression offloading.

A first glance at the `nvcomp` documentation might cause one to conclude that it does not support ZLIB. Whereas up till now we referred to ZLIB as an compression algorithm, this is in actuality, a CPU decompression library that uses DEFLATE as its underlying compression algorithm. As `nvcomp` does support DEFLATE, it also supports ZLIB(86).

The `nvcomp` library offers a so-called high-level decompression API, to which one can simply pass a buffer of compressed data, a decompression configuration object and a buffer for the decompressed data, to perform the decompression(87). `nvcomp` can derive this decompression configuration object from the compressed buffer, or the compression configuration object, that one creates during `nvcomp` compression. We want to evaluate the GPU decompression performance, for buffers of compressed pages, that do not originate from `nvcomp`. As such, we do not have these compression configuration objects. The alternative wherein we let the high-level API derive the decompression configuration object from the compressed buffer is also not feasible, because this assumes we store compressed data in the HLIF format, which is yet again only the case for buffers compressed using `nvcomp` (88).

Therefore, we use the low-level decompression API, that decompresses a batch of compressed chunks in parallel, after given an array of pointers to chunks, their compressed sizes, uncompressed sizes, and an array of pointers to memory that can store the decompressed

chunks(89). Note that we use NVIDIA terminology. We specifically, want to decompress a buffer of pages, whereby one can see a buffer as a batch, and each page in it, as a chunk.

5.3.2 Modelling $T_{\text{decompress}}$

Apart from outlining the inner workings of our reference GPU decompression library, we have not yet defined the way in which we actually model $T_{\text{decompress}}$ for an RDataFrame program, a certain dataset type and a compressed buffer of size C. In this context, we define $T_{\text{decompress}}$ as follows.

$$T_{\text{decompress}} = \frac{\left\lfloor S_{\text{processed}}^{\text{compressed}} / C \right\rfloor \cdot C \cdot r}{\lambda_{\text{decompress}}^{\text{GPU}}(C)} + \frac{S_{\text{produced}}^{\text{uncompressed}} - \left\lfloor S_{\text{processed}}^{\text{compressed}} / C \right\rfloor \cdot C \cdot r}{\lambda_{\text{decompress}}^{\text{GPU}}(S_{\text{processed}}^{\text{compressed}} \% C)}$$

This equation resembles that of the buffered page transfer time, defined previously. Its leftmost term defines the time spent on decompressing full compressed buffers, whereas the rightmost term defines the time spent on decompressing the remainder of all compressed buffers. In each term, the numerator describes the amount of uncompressed data that an RDataFrame application produces. Dividing by the speed at which the GPU produces this data, yields the time of decompressing full buffers and their remainder, respectively. We define the speed at which the GPU produces uncompressed data, as a function of the buffer size C, because this directly influences the amount of exploitable parallelism. Given that the first term concerns the decompression of full buffers, we use C as an argument. On the other hand, the second term describes the decompression of the remaining compressed data. As such we pass the decompression speed function, the size of this remaining compressed data.

5.3.3 Determining $\lambda_{\text{decompress}}^{\text{GPU}}$

For our NVIDIA A6000 GPU, and each aforementioned compression algorithm, we want to know a sensible buffer size value, that maximises the decompression speeds for the LHCB benchmark. Therefore we will determine the function $\lambda_{\text{decompress}}^{\text{GPU}}$ for this application, for all supported compression algorithms. We also need these functions, to make actual $T_{\text{decompress}}$ predictions.

In order to determine $\lambda_{\text{decompress}}^{\text{GPU}}(C)$ for a dataset, we do the following. For starters, we write all used compressed pages to separate files, retaining the order in which the single-threaded RDataFrame framework decompresses them, through their names. We insert code that does this, in the `PopulatePageFromCluster` method of `RPageStorageFile`, because page decompressions occur there (82).

After this, we allocate compressed pages to buffers of size C, with a program called `combine`, that we wrote ourselves. This program allocates pages to buffers in the order that the single-threaded RDataFrame program encounters them for decompression, to keep their contents realistic with respect to a concrete GPU offloading implementation. It often occurs that C does not perfectly divide the combined size of all used compressed pages, of a dataset. In the last created buffer, we therefore loop back to the first compressed page, to re-start allocation from this page and onwards, potentially looping back multiple times, until the last buffer is full as well.

We then pass each buffer to a program that invokes the low-level `nvcomp` API to decompress it, while also measuring the average decompression time (n=100) with 100 warmup decompressions for the same buffer. That is, we measure the average latency of the CUDA decompression function, after creating, allocating and transferring all required data to the GPU. The program also prints the amount of compressed data present in the buffer on which we invoke it. Jolly Chen originally developed this program, whereas we made some modifications on which we will elaborate later (90).

Now that we know the average decompression time for each buffer of size C, corresponding to a dataset, as well as the amount of compressed data in each buffer, we can sum the amounts of compressed data. Then, we can compute the amount of uncompressed data, using the compression ratio of all used columns, of the dataset from which we created the buffers. Finally, dividing by the sum of the the decompression time for each buffer, yields the speed $\lambda^{\text{GPU}}_{\text{decompress}}(C)$ at which the GPU produces uncompressed data, for a certain dataset, when creating buffers of size C. Repeating this procedure for various values of C, allows us to find the relationship between C and $\lambda^{\text{GPU}}_{\text{decompress}}$ for a specific dataset.

5.3.4 $\lambda^{\text{GPU}}_{\text{decompress}}$ for ZSTD

One can use this procedure to find the relationship between C and $\lambda^{\text{GPU}}_{\text{decompress}}(C)$ for any dataset RDataFrame program combination. We will however, determine relationships for datasets of the B2HHH type, belonging to the LHCB program. More specifically, we will obtain a separate relationship for ZLIB, ZSTD, and LZ4. We can not do so for LZMA, because `nvcomp` lacks support for this algorithm.

In the context of an compression algorithm, previous statements assume that there is no difference in its relationship, across datasets of the B2HHH type. We will check this assumption by determining $\lambda^{\text{GPU}}_{\text{decompress}}(C)$ for all single-threaded calibration/verification sets compressed with ZSTD, where C ranges between 4 times the maximum uncompressed used page size for a dataset, and 2000 MB.

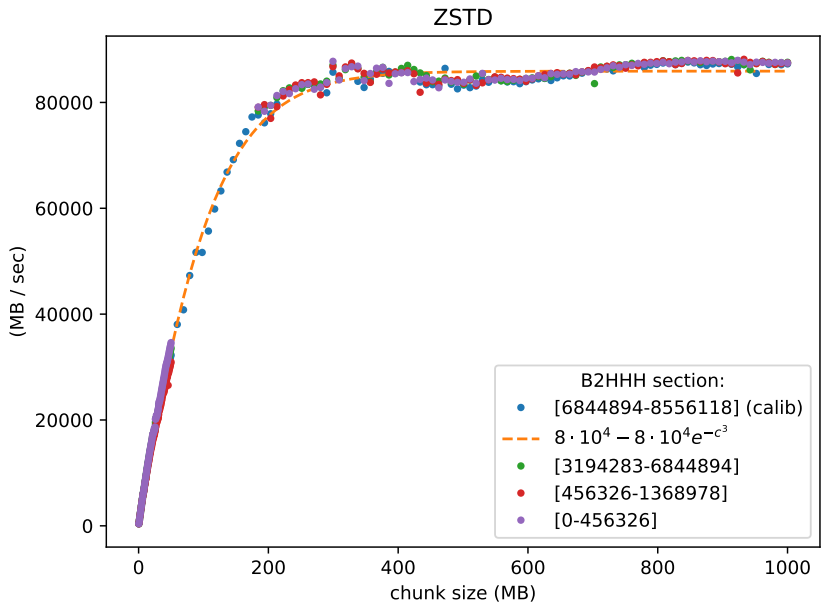


Figure 5.9: Each set of colored points displays the average speeds at which the nvcomp low-level API produces uncompressed data across chunks of various sizes, produced from the used pages of a single-threaded ZSTD calibration/verification dataset. We took these measurements on a NVIDIA A6000 GPU, embedded in a DAS6 node. The dotted line is a function fit of all points corresponding to the single-threaded ZSTD calibration set.

Figure 5.9 shows $\lambda^{\text{GPU}}_{\text{decompress}}(C)$ for each ZSTD dataset from the single-threaded calibration/verification, as a separate color. Unfortunately, for all non-calibration datasets, a measurement error causes missing points between approximately 50 MB and 185 MB. Nevertheless, visually, the closeness of all other points across the various datasets, provides evidence for the assumption that $\lambda^{\text{GPU}}_{\text{decompress}}(C)$ remains the same across LHCB datasets of the same type.

Apart from this, we can see that the speed at which ZSTD produces uncompressed data, gradually increases with the chunk size, up till approximately 85910.9 MB/sec. All computed speeds are bigger than 95% of this value, from a chunk size of 232 MB and onwards. Dividing the maximum speed at which the GPU produces uncompressed ZSTD data, by the average compression ratio of 1.45 (var=4.14e-7) across all used columns of ZSTD single-threaded verification/calibration sets, yields the speed at which the GPU processes compressed ZSTD data, which is 59248.9 MB/sec.

The figure also shows a fitted function for the ZSTD calibration set, whose form $c - \beta \cdot c \cdot e^{-bC^d}$ is a slight modification of the expression we used when discussing buffered data transfers. More specifically, we added a parameter as a power of C, the chunk

size. Yet again, we perform fitting using `curve_fit` from the `scipy` library. Due to space constraints the figure does not display the model parameters fully, whom are: $c = 8.59 \cdot 10^4$, $\beta = 9.75 \cdot 10^{-1}$, $b = 5.08 \cdot 10^{-3}$ and $d = 1.15$. When restricting chunk sizes to those equal and above 5.3 MB, the model deviates at least -6.6% and at most 8.1% from the calibration set. Without this restriction, the upper bound on deviations increases to 507.2%.

As such, one can use this model of $\lambda_{\text{GPU}}^{\text{decompress}}$ to make reasonably accurate $T_{\text{decompress}}$ predictions whenever the chunk size C and the remainder $S_{\text{processed}}^{\text{compressed}} \% C$ are higher than 5.3 MB. If we want to predict $T_{\text{decompress}}$ as a function of $S_{\text{processed}}^{\text{compressed}}$, and set our chunk size to a performance-wise sensible 232 MB, we satisfy the first criterion. For second criterion, we first and foremost know that $S_{\text{processed}}^{\text{compressed}} \% C$ cycles between 0 and C for increasing $S_{\text{processed}}^{\text{compressed}}$, as exemplified by figure 5.10. This figure also shows, that $S_{\text{processed}}^{\text{compressed}} \% C$ is only smaller than 5.3 MB, for the first part of each cycle. That is only $(5.3/232) \cdot 100\% = 2\%$ of a cycle, and thus only 2% of the entire domain of $S_{\text{processed}}^{\text{compressed}}$. In accordance with the second criterion, we thus only get inaccurate $T_{\text{decompress}}$ predictions, for 2% of the domain of $S_{\text{processed}}^{\text{compressed}}$ when using our current estimation of $\lambda_{\text{GPU}}^{\text{decompress}}$. The impact of the inaccuracy of $\lambda_{\text{GPU}}^{\text{decompress}}$ on predictions of $T_{\text{decompress}}$ is thus relatively limited.

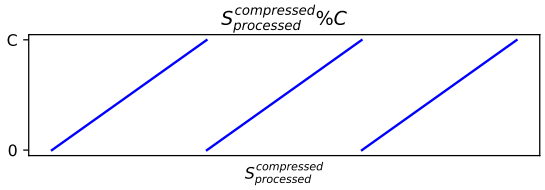


Figure 5.10: Displays $S_{\text{processed}}^{\text{compressed}} \% C$ with respect to $S_{\text{processed}}^{\text{compressed}}$.

5.3.5 $\lambda_{\text{decompress}}^{\text{GPU}}$ for LZ4 and ZLIB

Now we will proceed and determine $\lambda_{\text{decompress}}^{\text{GPU}}$ for LZ4 and ZLIB, in the context of the LHCB benchmark. Unfortunately, we can not use the same approach as before. For some reason, the low-level API of `nvcomp` produces an error, when fed with a buffer of pages from either algorithms, originating from a single-threaded verification dataset. Most likely, this occurs because there is a format mismatch between pages, that one compresses using ROOT, and `nvcomp`, even though the compression algorithms do not differ.

Because we still wanted to obtain an estimate of $\lambda_{\text{decompress}}^{\text{GPU}}$ for either algorithms, we decided to obtain all used uncompressed pages from the smallest LZ4 and ZLIB single-threaded verification set, and compress them ourselves, using `nvcomp`. We use the smallest

single-threaded verification dataset for LZ4 and ZLIB, because this minimises the time required to obtain all data to find both $\lambda^{\text{GPU}}_{\text{decompress}}$ relations. Supporting this decision is our previous analysis, which shows that $\lambda^{\text{GPU}}_{\text{decompress}}$ does not vary across the different single-threaded ZSTD verification/calibration sets.

5.3.5.1 Extracting the uncompressed pages

To extract all used uncompressed pages from either dataset, we again added code within the `PopulatePageFromCluster` method of the `RPageSourceFile` instantiation (82). After decompressing a page, we write its uncompressed variant to disk. We also save the order in which we encounter the pages, for the construction of buffers of compressed pages, at a later point in time. Recompiling the framework, as well as the LHCB program, and applying it to the LZ4 and ZLIB single-threaded verification set, then yields all uncompressed pages which we need.

5.3.5.2 Compressing pages with nvcomp

To compress the pages of either dataset using `nvcomp`, we wrote a program based on the low-level API example by NVIDIA (91). This program reads all uncompressed pages and places them in a buffer. Apart from this, it also creates an array of pointers to all pages in this buffer, and allocates a buffer for each compressed page counterpart. Given this, the program then invokes the low-level compression API.

After checking the compression status that can indicate potential errors that can occur, it decompresses the compressed data, to check whether the resulting uncompressed buffer differs from the original uncompressed buffer. Apart from this, we also check the global decompression status object, and those corresponding to each individual page.

Finally, we write the compressed pages to disk. More specifically, we write a separate file per compressed page. For each page, we again store the order in which an RDataFrame program encounters it, through the filename, and we also store its uncompressed size, in this filename. We store the uncompressed size of each used page, because the program that measures the latency of decompressing buffers, requires this information. For each single-threaded ZSTD calibration/verification dataset, ROOT embeds this info within a compressed page header, so that is why we did not need to keep track of this information previously.

5.3.5.3 Circumventing ZLIB chunk size limitations in nvcomp

While attempting to compress all used uncompressed ZLIB pages, corresponding to their single-threaded verification set, we found out that `nvcomp` imposes a maximum size on each chunk that it can compress. More specifically, we found that some pages of the smallest single-threaded verification set, exceed the 65536 bytes size limit of ZLIB pages. As such we split all uncompressed ZLIB pages of size M, that exceed this size, in $1 + \lfloor (M-1)/65536 \rfloor$ pages of size 65536 bytes and one remainder page of $M\%65536$ bytes.

Remember that we number all pages. Imagine that a page with index i has a size bigger than 65536 bytes. We then index the newly created pages from i, whereby we imaginatively start from the front of the original uncompressed page, and assign indexes until we reach the last splitted descendant. Additionally, we ensure that all pages originally indexed with $i + 1$ and onwards, shift with the amount of newly created splitted pages. With this methodology, we can maintain the order of data across the different compressed buffers. Now that we have a set of uncompressed pages, corresponding to the used pages of the smallest single-threaded ZLIB verification set, which does satisfy the `nvcomp` size criterion, we compress these pages with the low-level `nvcomp` compression API, using ZLIB algorithm 1.

5.3.5.4 Modifications of combine and the program by Jolly Chen

From this point onward, we can approximately use the same methodology as we did for ZSTD, to obtain $\lambda^{\text{GPU}}_{\text{decompress}}$ for ZLIB and LZ4. Namely, for each algorithm, we create compressed buffers for various values of C using `combine`, which we then pass to the program, that invokes the low-level decompression API and also measures the average decompression latency. We equate both processes approximately, because they differ in one aspect.

This difference occurs, because compressed pages do not longer contain a header that stores their uncompressed and compressed size. Because the decompression latency measurement program by Jolly Chen, requires this info, and expects these page headers, we modified it so that it does not necessarily require this header, and that it can read the uncompressed and compressed sizes from two meta-data files, that accompany a compressed buffer file. As such, we also modified the `combine` program so that it can output these meta-data files, for a compressed buffer. The decompression latency measurement program requires this info, to set up an array of pointers to all pages in the compressed buffer for which it performs measurements, and to allocate and create pointers to the space for

the decompressed page counterparts. It feeds these objects, as well as the uncompressed and compressed page sizes, into the `nvcomp` low-level decompression API, to start the decompression.

5.3.5.5 Decompression speed results

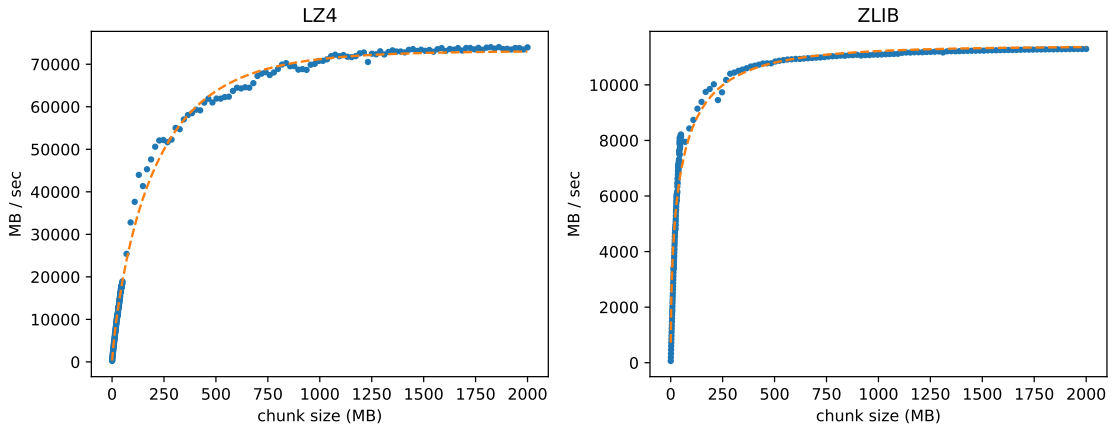


Figure 5.11: The dotted points of both graphs display the relationship between the chunk size and the measured decompression speeds using nvcomp for the smallest ZLIB and LZ4 single-threaded LHCB verification dataset. We compressed these datasets ourselves using nvcomp. We took all measurements on a DAS6 node. See table 7.1 for specifics. Both orange dotted lines are function fits of $c - \beta \cdot c \cdot e^{-b \cdot C^d}$ to measurements. For LZ4 the exact parameters are $c = 7.31 \cdot 10^4, \beta = 1, b = 1.15 \cdot 10^{-2}$ and $d = 8.34 \cdot 10^{-1}$. On the other hand, for ZLIB they are $c = 1.14 \cdot 10^4, \beta = 1, b = 1.32 \cdot 10^{-1}$ and $d = 5.0 \cdot 10^{-1}$.

Figure 5.11 shows the speed at which the NVIDIA A6000 GPU produces uncompressed data for the smallest LZ4 and ZLIB single-threaded verification dataset, across various chunk sizes. From the graphs in figure 5.11, we can see that the speeds grow, when increasing the chunk size. From the data underlying these plots, we know that the GPU reaches a speed of approximately 73081.4 MB/sec for uncompressed LZ4 data. For all chunk sizes at least as big as 956 MB, we reach 95% of this speed. In the same way, we know that the GPU reaches a speed of approximately 11390.7 MB/sec for uncompressed ZLIB data. We reach 95% of this speed, for all chunk sizes bigger than or equal to 503 MB.

Because we compressed the ZLIB and LZ4 dataset using `nvcomp`, we use the compression ratio's obtained by `nvcomp` to compute all aforementioned speeds and those shown in figure 5.11, rather than those of the original datasets. These `nvcomp` compression ratio's equal 1.29 and 1.50, for LZ4 and ZLIB, respectively. Given these compression ratio's, we can

also compute the speed at which the GPU processes compressed data, for the smallest LZ4 and ZLIB single-threaded verification dataset. Those equal 56652.2 MB/sec and 7593.8 MB/sec, for LZ4 and ZLIB, respectively.

As we previously did in determining $\lambda_{\text{GPU}}^{\text{decompress}}$ for ZSTD, we also fitted the expression $c - \beta \cdot c \cdot e^{-bC^d}$ to the data shown in figure 5.11. Such a fitted function permits one to predict $T_{\text{decompress}}$ for ZLIB and LZ4. The fitted functions estimate the measured decompression speeds well, for chunk sizes above 148 MB and 69 MB, for LZ4 and ZLIB, respectively. With this we mean that their deviations only lie between -10% and 10% of the measured decompression speeds, for chunk sizes above these values. Using the same rationale as for ZSTD, we know that with a chunk size set at 956 MB and 503 MB for LZ4 and ZLIB, respectively, we can make accurate predictions of $T_{\text{decompress}}$ for most of the domain of $S_{\text{processed}}^{\text{compressed}}$. More specifically, this is approximately 85% for both algorithms.

5.4 Predicted GPU decompression speedups

5.4.1 Full speedup prediction model

Now that we can model buffered page transfer times, the GPU decompression time and the multi-threaded CPU decompression time, we can predict the speedup of the corresponding GPU decompression offloading implementation, with respect to the current CPU decompression implementation. The following equation defines this speedup.

$$\text{Speedup} = \frac{T_{\text{decompress}}^{\text{multithread}}(S)}{T_{\text{D2H}} + T_{\text{decompress}} + T_{\text{H2D}}} \tag{5.19}$$

For all terms in this equations, we recap all equations that we defined previously, below.

$$T_{\text{decompress}}^{\text{multithread}}(S) = \frac{1}{N_{\text{threads}}} \sum_{q \in \mathcal{F}} \left\lceil \frac{S_q^{\text{uncompressed}} \cdot f_q^{\text{decompress}}}{\lambda^{t(q),C(q),L(q)}} \right\rceil$$

$$T_{\text{H2D}} = \frac{\lfloor S_{\text{transferred}}^{\text{compressed}} / C \rfloor \cdot C}{\lambda^{\text{H2D}}(C)} + \frac{S_{\text{transferred}}^{\text{compressed}} \% C}{\lambda^{\text{H2D}}(S_{\text{transferred}}^{\text{compressed}} \% C)}$$

$$T_{\text{decompress}} = \frac{\left\lfloor S_{\text{processed}}^{\text{compressed}} / C \right\rfloor \cdot C \cdot r}{\lambda_{\text{decompress}}^{\text{GPU}}(C)} + \frac{S_{\text{produced}}^{\text{uncompressed}} - \left\lfloor S_{\text{processed}}^{\text{compressed}} / C \right\rfloor \cdot C \cdot r}{\lambda_{\text{decompress}}^{\text{GPU}}(S_{\text{processed}}^{\text{compressed}} \% C)}$$

$$T_{\text{D2H}} = \frac{\lfloor S_{\text{transferred}}^{\text{compressed}} / C \rfloor \cdot C \cdot r}{\lambda^{\text{D2H}}(C \cdot r)} + \frac{S_{\text{transferred}}^{\text{uncompressed}} - \lfloor S_{\text{transferred}}^{\text{compressed}} / C \rfloor \cdot C \cdot r}{\lambda^{\text{D2H}}(S_{\text{transferred}}^{\text{uncompressed}} - \lfloor S_{\text{transferred}}^{\text{compressed}} / C \rfloor \cdot C \cdot r)}$$

We want to show how to utilise equation 5.19, to make speedup predictions for an RDataFrame program applied on a dataset of a specific type, where a certain compression algorithm is

used, across various thread counts and dataset sizes. Obviously, we make these predictions for a specific node, consisting of a CPU and a GPU. We also fix the compressed buffer size C in these predictions.

Apart from discussing this process in general terms, we will demonstrate it for the LHCB benchmark. More specifically, we will demonstrate this for datasets of the B2HHH type belonging to this program, for some of the supported compression algorithms. We only look at LZ4, ZLIB and ZSTD, because `nvcomp` does not support LZMA. Regarding the hardware, we assume usage of a hypothetical node with an AMD 7220P, and a NVIDIA A6000 GPU, whom are linked by PCIE 4.0x16 trough DDR4 main memory. Additionally, we fix C for each compression algorithm/level so that we reach 95% of transfer speeds and GPU decompression speed. For LZ4, ZLIB and ZSTD we set C at 956 MB, 503 MB and 232 MB, respectively.

5.4.1.1 Restricting dataset sizes and thread counts with $T^{\text{multithread}}_{\text{decompress}}$

Remember that for an RDataFrame program, we assume that given a certain compression algorithm/level, there is a type of dataset that is meant to be used with this program. For datasets of this type and specific hardware, we can predict $T^{\text{multithread}}_{\text{decompress}}$ with the formulated model accurately, across a range of thread counts, and for each thread count, a range of dataset sizes. It is important to know these parameter ranges, as this limits the range for which we can make accurate GPU decompression offloading speedup predictions.

For the LHCB benchmark, the single-threaded verification showed that the calibrated model estimates the decompression latency well across all the B2HHH dataset sections, regardless of the used compression algorithm/level. The smallest section consisted of 456326 events. We will assume that the model is accurate for the LHCB benchmark applied on any dataset of the B2HHH type, that consists of more than 456326 events, even when containing more events than the biggest section that consists of 3650611 events. With $N_{\text{thread}} = 1$, we will thus only make GPU offloading speedup predictions, for the LHCB benchmark applied on datasets of the B2HHH type, for $N_{\text{events}} > 456326$. The multi-threaded model verification proved that we can make accurate LHCB benchmark latency predictions for the thread count range of 1-21, for LZ4, ZLIB and ZSTD, with a B2HHH dataset containing more than 68448940 events. We will thus restrict the thread count range for which we make GPU offloading speedup predictions with respect to the multi-threaded LHCB benchmark, uptill 21 threads, and we will only consider B2HHH dataset sizes bigger than 68448940 events.

5.4.1.2 Expressing dataset sizes in N_{events}

It is intuitive to make speedup predictions across values of a single variable that quantifies the dataset size. Currently, we have many variables for this, across all equations that define the T terms in equation 5.19. For the LHCB benchmark, we can easily re-express these variables in terms of the amount of events N_{events} in a dataset. We will start doing so now.

In the previous discussion, we already limited the dataset sizes, for which we compute the GPU offloading speedup, in terms of N_{events}. These ranges are equal to those where $T_{\text{decompress}}^{\text{multithread}}$ is accurate. In chapter 2, we derived the following expression for $T_{\text{decompress}}^{\text{multithread}}(N_{\text{events}})$ when we apply the LHCB benchmark on datasets of the B2HHH type.

$$T_{\text{decompress}}^{\text{multithread}}(N_{\text{events}}) = \frac{1}{N_{\text{threads}}} \left(15 \cdot \frac{N_{\text{events}} \cdot 8}{\lambda^{\text{SplitReal64}}} + 3 \cdot \frac{N_{\text{events}} \cdot 4}{\lambda^{\text{SplitInt32}}} \right)$$

From the single-threaded decompression model calibration, we know the CPU decompression speeds λ on an `alma9` node, for the LHCB benchmark applied on datasets of the B2HHH type, that use ZLIB, ZSTD or LZ4 compression. One can find these values in table 5.2.

	$\lambda^{\textbf{SplitInt32}}$	$\lambda^{\textbf{SplitReal64}}$
ZLIB	0.82 GB/sec	0.32 GB/sec
ZSTD	2.25 GB/sec	2.32 GB/sec
LZ4	3.14 GB/sec	5.57 GB/sec

Table 5.2: The calibrated single-threaded model parameters, for the LHCB benchmark applied on B2HHH datasets, across different compression algorithms, when using an alma9 node.

In all other equations defining T variables, we currently express the dataset size in variables $S_{\text{transferred}}^{\text{compressed}}$, $S_{\text{processed}}^{\text{compressed}}$, $S_{\text{transferred}}^{\text{uncompressed}}$, and $S_{\text{produced}}^{\text{uncompressed}}$. We will now start with rewriting these in terms of N_{events}. First of all note that $S_{\text{transferred}}^{\text{compressed}} = S_{\text{processed}}^{\text{compressed}}$, and $S_{\text{transferred}}^{\text{uncompressed}} = S_{\text{produced}}^{\text{uncompressed}}$. Because the LHCB benchmark does not skip page decompression's or need multiple decompression's for a page, the compressed data size that the LHCB benchmark processes $S_{\text{processed}}^{\text{compressed}}$, equals the compressed size of all used dataset columns $S^{\text{compressed}}$. Due to the same argument, the uncompressed data size that the LHCB benchmark produces $S_{\text{produced}}^{\text{uncompressed}}$ equals the uncompressed size of all used dataset columns $S^{\text{uncompressed}}$. We can relate $S^{\text{uncompressed}}$ and $S^{\text{compressed}}$ with the compression ratio r of all used columns. Additionally, we know that the LHCB benchmark uses 18 columns for each event, whereby 15 columns contain values with a size of 8 bytes and 3 columns contain

values with a size of 4 bytes. We can thus state that $S^{\text{uncompressed}} = N_{\text{events}} \cdot (15 \cdot 8 + 3 \cdot 4)$. All in all, we can thus rewrite the variables that define dataset sizes in all T equations, for the LHCB benchmark applied on B2HHH datasets, as follows.

$$S^{\text{compressed}}_{\text{transferred}} = S^{\text{compressed}}_{\text{processed}} = (N_{\text{events}} \cdot (15 \cdot 8 + 3 \cdot 4))/r$$
$$S^{\text{uncompressed}}_{\text{produced}} = S^{\text{uncompressed}}_{\text{transferred}} = N_{\text{events}} \cdot (15 \cdot 8 + 3 \cdot 4)$$

5.4.1.3 Setting the compression ratio's

The expressions defining each T variable in equation 5.19, use the compression ratio r. For the LHCB benchmark specifically, we also define $S^{\text{compressed}}_{\text{transferred}}$ in terms of r. For ZSTD, ZLIB and LZ4, we calibrated the multi-threaded CPU decompression model, for the LHCB program applied on a B2HHH dataset, with all used columns considered in unison, having a compression ratio of 1.46, 1.45 and 1.41, respectively. On a `das6` node with an NVIDIA A6000 GPU, we determined $\lambda^{\text{decompress}}_{\text{GPU}}$ for ZSTD, on each of the original single-threaded calibration/verification LHCB datasets. Refer to section 5.3.4 for its equation and parameters. For LZ4 and ZLIB, we determined $\lambda^{\text{decompress}}_{\text{GPU}}$ for the smallest single-threaded LHCB verification set that we compressed ourselves using `nvcomp`, realising a compression ratio of 1.29 and 1.50, respectively. See the caption of figure 5.11 for its equation and the parameters.

When making speedup predictions for each of the considered compression algorithms, across various N_{event} and thread count values, we fix r. For this we assume that the dataset size does not influence r. In the context of all original single-threaded calibration/verification B2HHH dataset sections, we have some evidence for this assumption, as the variance of all compression ratio's is in the order of 10^{-7}.

Remember that for LZ4 and ZLIB, we can choose between the compression ratio's achieved using `nvcomp` and those of the original datasets, compressed by ROOT. Because we want to predict the speedup for the LHCB benchmark applied on real B2HHH datasets, we use the original compression ratio's. Regardless of the considered compression algorithm/level, we can only use the found $\lambda^{\text{decompress}}_{\text{GPU}}$ functions. For LZ4 and ZLIB, this assumes that the functions $\lambda^{\text{decompress}}_{\text{GPU}}$ do not change due to the difference in the obtained compression ratio with ROOT and `nvcomp`. We also assume that the functions $\lambda^{\text{decompress}}_{\text{GPU}}$ for each considered compression algorithm/level, remain constant across datasets of the B2HHH type with different sizes. Evidence for this assumption are the visually similar relations for different B2HHH dataset sections for the ZSTD compression algorithm shown in figure 5.9.

5.4.1.4 Considering the impact of inaccuracies in $\lambda_{\mathbf{H2D}}$, $\lambda_{\mathbf{D2H}}$ and $\lambda_{\mathbf{GPU}}^{\mathbf{decompress}}$

When making speedup predictions for varying thread counts and dataset sizes, it is important to consider the impact of inaccuracies in the models λ_{H2D}, λ_{D2H} and $\lambda_{\text{GPU}}^{\text{decompress}}$. For our hardware we know that λ_{H2D} and λ_{D2H} have a high deviation (outside the $\pm$ 10% bound) for a transfer size of 0.001 MB. Note that one can find the equation and parameters of these functions, in table 5.1. We ignore the high error because it will only impact the transfer time of a very small remainder compressed buffer or its decompressed counterpart, which inherently contributes very little to T_{D2H} and T_{H2D}.

When evaluating the inaccuracy of $\lambda_{\text{GPU}}^{\text{decompress}}$ for ZLIB, LZ4 and ZSTD, on datasets of the B2HHH type, we notice that they all have a certain lower bound D from which they provide accurate decompression speeds. In our case, the full compressed buffers always have a size above this lower bound. This is not always the case for the remainder. Remember that $S_{\text{processed}}^{\text{compressed}} \% C$ cycles between 0 and C. Inaccurate values of $\lambda_{\text{GPU}}^{\text{decompress}}(S_{\text{processed}}^{\text{compressed}} \% C)$ only occur when $S_{\text{processed}}^{\text{compressed}} \% C \leq D$, which is the case if and only if $S_{\text{processed}}^{\text{compressed}} \in [C \cdot Q, C \cdot Q + D]$ with $Q \in \mathbb{N} \cup \{0\}$. Using the definition of $S_{\text{processed}}^{\text{compressed}}$, we can rewrite this range in terms of N_{events} and r. In other words, $\lambda_{\text{GPU}}^{\text{decompress}}$ produces inaccurate predictions for $N_{\text{events}} \in [(C \cdot Q \cdot r)/132, ((C \cdot Q + D) \cdot r)/132]$. When making speedup predictions for the LHCB benchmark applied on a B2HHH dataset where its used columns have a compression ratio r, we now know the ranges of dataset sizes wherein we get inaccurate predictions.

5.4.1.5 Single-threaded speedup predictions

Figure 5.12 shows the GPU decompression offloading speedup predictions on our hardware, with respect to single-threaded execution of the LHCB benchmark, applied on B2HHH datasets of various sizes, compressed using ZLIB, LZ4 or ZSTD. All red regions indicate areas where predictions suffer from the inaccuracies of $\lambda_{\text{decompress}}^{\text{GPU}}$.

As one can see, all computed speedups are higher than one meaning that GPU decompression offloading is beneficial in this context. For each compression algorithm, the speedup is the lowest for the smallest N_{events} values. When we evaluate the speedups for increasing N_{events} values, it grows until we reach maximum, after which it drops down to a lower value, from which we climb towards this maximum again. This growing, falling and climbing behavior continues to occur when looking at increasing N_{event} values, only for the lower value to which we drop to grow closer and closer towards this maximum.

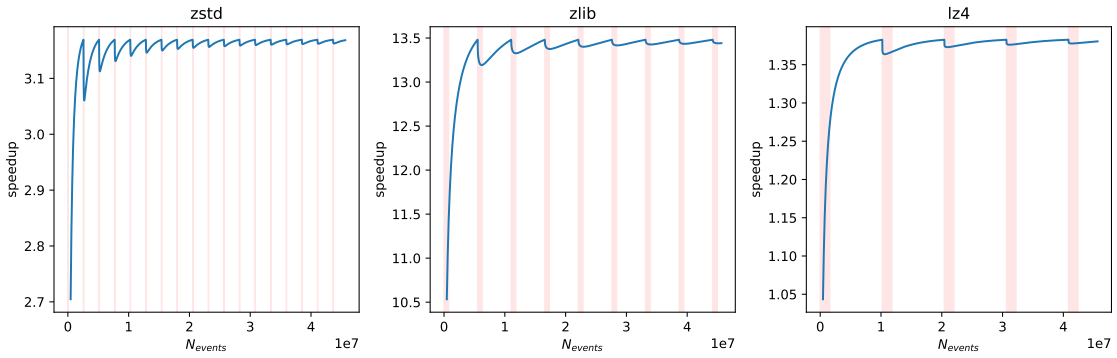


Figure 5.12: Shows the GPU decompression offloading speedup prediction with respect to single-threaded execution of the LHCB benchmark, for different sizes of the B2HHH dataset, across different compression algorithms. We computed these speedups for a hypothetical node with an AMD EPYC 7702P processor and a NVIDIA A6000 GPU. In all red areas, there are inaccurate speedup predictions due to the inaccuracy of $\lambda^{\text{GPU}}_{\text{decompress}}$.

We know that the single-threaded CPU decompression time grows linearly with respect to the dataset size N_{events}. So this behavior must have something to do with the GPU implementation. In this GPU decompression implementation idea, we transfer the compressed data to the GPU and decompress it there, through full compressed buffers and a buffer with the remaining compressed data. For each of those compressed buffers we transfer a decompressed counterpart back to the CPU.

In each graph in figure 5.12, we start at a dataset size which does not fill the compressed buffer of size C. Even though we have a partially filled compressed buffer, we can transfer it, as well as its decompressed counterpart, at full speed on our considered hardware. That is because for the smallest considered dataset size, all used columns have a uncompressed size of 60 MB, and a compressed size of approximately 40 MB (slightly varying over different compression algorithms). These sizes lie greatly above 21 MB, the size for which we reach 95% of our maximum D2H and D2H speeds. We only reach maximum GPU decompression speeds for a compressed buffer of size C, and initially, we are not at that point for any of the compression algorithms. Increasing the dataset size, allows us to gain a higher GPU decompression speed, which makes the GPU decompression offloading speedup grow bigger. This GPU decompression offloading speedup reaches its first maximum, with a full compressed buffer. Increasing the dataset size beyond this, introduces a new compressed buffer, apart from the one already in existence, that is only partially filled with data. For dataset sizes slightly above the size which reach the maximum speedup, the sub-optimal transfer speeds of the remainder compressed buffer as well as its decompressed counterpart,

in addition to its sub-optimal decompression speed, causes the speedup to drop. It does not drop as low as it was before, because a portion of the dataset is transferred and decompressed at full speed. When increasing the dataset size further, the new compressed buffer becomes fuller, and after it and its decompressed counterpart surpasses 21 MB, we maximize the transfer speeds of this new buffer. Only when the new compressed buffer reaches size C, we know that the decompression speed is at its peak as well, causing the maximum speedup to be reached again. This explanation covers all the observable falling, rising and climbing behavior.

We have not yet clearly elaborated on the reason why the value to which we drop, grows closer and closer to the maximum speedup. That is because the processing time of the remaining compressed buffer on the GPU, which fluctuates with respect to the dataset size, becomes less and less significant with respect to the processing time of full compressed buffers.

Another observation in these three graphs is that the cycle period grows from the leftmost graph to the rightmost graph. The cycle period is the amount of events, between two consecutive maxima of GPU decompression offloading speedups. At this point we know that we reach a maximum whenever we only need to process full compressed buffers of size C. Two consecutive maxima occur for a dataset that can fit in n full compressed buffers with a total size of $n \cdot C$ and $n + 1$ full compressed buffers with a total size of $(n + 1) \cdot C$. We can express these sizes in events using the compression ratio, and the size of an event which is 132 bytes in our case. This yields $(n \cdot C \cdot r)/132$ and $((n + 1) \cdot C \cdot r)/132$. The difference between those event counts is $(C \cdot r)/132$. We now have an equation governing the cycle period, expressed in events for datasets of the B2HHH dataset. Because we know that r is relatively similar across the three compression algorithms, the cycle period grows from left to right, because the compressed buffer size increases in the same way.

5.4.1.6 Multi-threaded speedup predictions

For B2HHH datasets compressed using ZLIB, we present the GPU decompression offloading speedup on our hardware with respect to multi-threaded decompression in the LHCB benchmark, across various thread counts and dataset sizes, in figure 5.13. As one can see, each graph in this figure displays the relationship between the predicted speedup and the B2HHH dataset size, for a specific thread count. In each graph, this speedup fluctuates as in the speedup graphs with respect to single-threaded decompression, in figure 5.12. This time, the fluctuation is smaller, because we start from a bigger dataset, wherein the

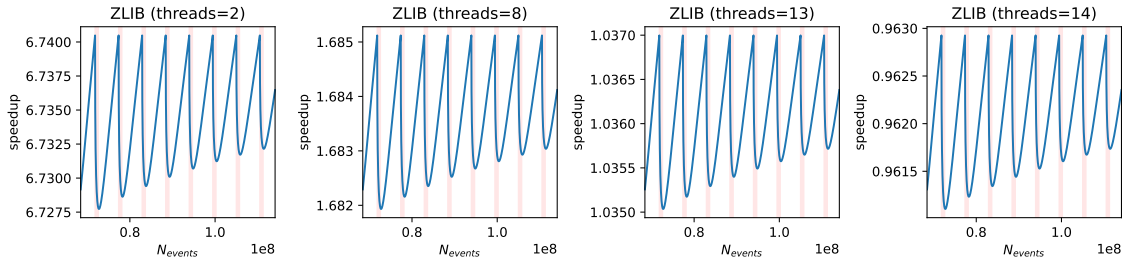


Figure 5.13: Displays the predicted GPU decompression offloading speedup with respect to the multi-threaded LHCB decompression time across various thread counts and sizes of a B2HHH dataset which uses ZLIB compression. Each graph displays the relation between dataset sizes and this predicted speedup, for a certain thread count. In each red area, inaccuracies in $\lambda^{\text{GPU}}_{\text{decompress}}$ cause inaccuracies in the speedup predictions.

processing time of the remainder compressed buffer is small with respect to that of all full compressed buffers together.

When evaluating all graphs in figure 5.13 from left to right, we can see that GPU decompression offloading for ZLIB is beneficial with respect to multi-threaded LHCB decompression, up till and including 13 threads. That is because only from 14 threads, the speedup drops below 1. We also created figures for the speedup with respect to multi-threaded decompression, for B2HHH datasets compressed with LZ4 and ZSTD. These figures 7.5 and 7.6 can be found in the appendix. From them we can conclude that GPU decompression offloading does not outperform LHCB decompression in a multi-threaded scenario for LZ4. Additionally, we can conclude that for ZSTD, GPU decompression offloading only outperforms LHCB decompression for two or three threads.

For each considered compression algorithm, the observations we make regarding the range of CPU threads that GPU decompression offloading can beat in terms of performance, are consistent with the conclusions we made by analysing buffered page transfer times in section 5.2.1.5. More specifically, for ZSTD and LZ4 we make exactly the same thread count range predictions. For ZLIB, we predict a potential performance improvement up till 21 threads based on analysing the buffered page transfer times, whereas in this speedup computation we predict a performance improvement up till 13 threads.

5.4.2 Maximum speedup prediction

In the previous section, we made speedup predictions for GPU offloading of decompression with respect to CPU decompression for the LHCB benchmark applied on B2HHH datasets compressed using various algorithms. We observed that whenever the compressed dataset

size is a multiple of C, we maximize the speedup. That occurs because we can then transfer the buffers at maximum speed, as well as decompress them on the GPU at maximum speed. When increasing the dataset size, we also converge towards this maximum, because the time for transferring and decompressing the remainder buffer on the GPU becomes less and less significant. Underlying these predictions is a model linear in the variable N_{events}, specifically calibrated for B2HHH datasets and our hardware, that predicts $T^{\text{multithread}}_{\text{decompress}}$. In these predictions we also use calibrated analytical expressions for $\lambda^{\text{H2D}}, \lambda^{\text{GPU}}_{\text{decompress}}$ and λ^{D2H}. These functions depend on hardware such as the GPU, the main memory, the CPU-GPU link, and the compressed buffer size. Additionally, $\lambda^{\text{GPU}}_{\text{decompress}}$ can vary over different types of datasets.

Under some conditions on $T^{\text{multithread}}_{\text{decompress}}, \lambda^{\text{H2D}}, \lambda^{\text{GPU}}_{\text{decompress}}$ and λ^{D2H}, we want to prove that the maximum speedup is equal to the following equation, which follows from equation 5.19, when we take the compressed dataset size as a multiple of C.

$$\max(\text{Speedup}) = \frac{T^{\text{multithread}}_{\text{decompress}}(S)}{\frac{S^{\text{compressed}}_{\text{processed}}}{\lambda^{\text{H2D}}(C)} + \frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{D2H}}(C \cdot r)}}$$

If we can express $T^{\text{multithread}}_{\text{decompress}}$ as $S^{\text{uncompressed}}_{\text{produced}} / (N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}})$, with $\lambda^{\text{CPU-unithread}}_{\text{decompress}}$ the single-threaded speed at which a CPU can produce uncompressed data for a specific dataset type, then it is possible to make max(Speedup) independent of the dataset size. We will show later how we can achieve this, but let us focus on the condition for now.

5.4.2.1 Modelling $T^{\text{multithread}}_{\text{decompress}}$ with one decompression speed

Remember that up till this point we model $T^{\text{multithread}}_{\text{decompres}}$ more granularly, with a separate decompression speed for each column type compression algorithm/level combination. Our new formulation of $T^{\text{multithread}}_{\text{decompress}}$ should create the same latency predictions, when varying the dataset size or the amount of threads, for a specific dataset type, and hardware. For this it is imperative that $\lambda^{\text{CPU-unithread}}_{\text{decompress}}$ remains constant in this context. We find some conditions for this constancy, by equating both expressions modelling $T^{\text{multithread}}_{\text{decompress}}$ and then

solving for $\lambda_{\text{decompress}}^{\text{CPU-unithread}}$.

$$\begin{aligned}
\frac{S_{\text{produced}}^{\text{uncompressed}}}{N_{\text{threads}} \cdot \lambda_{\text{decompress}}^{\text{CPU-unithread}}} &= \frac{1}{N_{\text{threads}}} \sum_{q \in \mathcal{F}} \frac{S_{\text{produced}-q}^{\text{uncompressed}} \cdot f_q^{\text{decompress}}}{\lambda^{t(q),C(q),L(q)}} \\
\frac{S_{\text{produced}}^{\text{uncompressed}}}{\lambda_{\text{decompress}}^{\text{CPU-unithread}}} &= \sum_{q \in \mathcal{F}} S_{\text{produced}-q}^{\text{uncompressed}} \cdot f_q^{\text{decompress}} \cdot \frac{1}{\lambda^{t(q),C(q),L(q)}} \\
\lambda_{\text{decompress}}^{\text{CPU-unithread}} &= \frac{S_{\text{produced}}^{\text{uncompressed}}}{\sum_{q \in \mathcal{F}} S_{\text{produced}-q}^{\text{uncompressed}} f_q^{\text{decompress}} \cdot \frac{1}{\lambda^{t(q),C(q),L(q)}}} \\
\lambda_{\text{decompress}}^{\text{CPU-unithread}} &= \frac{1}{\sum_{q \in \mathcal{F}} \frac{S_{\text{produced}-q}^{\text{uncompressed}} f_q^{\text{decompress}}}{S_{\text{produced}}^{\text{uncompressed}}} \cdot \frac{1}{\lambda^{t(q),C(q),L(q)}}}
\end{aligned} \tag{5.20}$$

From the final equation we can deduce that $\lambda_{\text{decompress}}^{\text{CPU-unithread}}$ remains constant for datasets of various sizes of a specific type as well as different thread count settings, whenever two conditions hold. First and foremost, the single-threaded decompression speeds $\lambda^{t(q),C(q),L(q)}$ should remain constant across various dataset sizes, and thread counts. Additionally, for each used column, the ratio of the amount of produced uncompressed data, with respect to the total amount of produced uncompressed data, should remain constant over all dataset sizes. When assuming that our single and multi-threaded decompression time models accurately model the decompression time, as we have shown for the LHCB benchmark, for some of the used B2HHH calibration/verification datasets, we satisfy the first criterion. A discussion with the CERN team revealed that the second condition holds for most RDataFrame applications.

We will now reason why the second condition holds for the LHCB benchmark applied on B2HHH datasets. For our single and multi-threaded calibration/verification B2HHH datasets, duplicate or page decompression skips do not occur, so we can rewrite the fraction $(S_{\text{produced}-q}^{\text{uncompressed}} f_q^{\text{decompress}})/S_{\text{produced}}^{\text{uncompressed}}$ as $S_q^{\text{uncompressed}}/S_{\text{produced}}^{\text{uncompressed}}$. The latter is the ratio of the uncompressed size of an used column, with respect to the uncompressed size of all used columns considered together. Given the LHCB benchmark and its B2HHH datasets, we know that we use 15 columns of the SplitReal64 type and 3 columns of the SplitInt32 type. We also know formulas for the uncompressed size of both column types, as well as for the total size of all used uncompressed columns considered together. Plugging these formulas in the fraction for both column types as shown below, allows us to conclude that the fraction does not depend on the dataset size, for each used column of B2HHH datasets.

$$\forall q \in \text{LHCB(B2HHH)} \frac{S_q^{\text{uncompressed}}}{S_{\text{produced}}^{\text{uncompressed}}} = \begin{cases} \frac{8 \cdot N_{\text{events}}}{15 \cdot 8 \cdot N_{\text{events}} + 3 \cdot 4 \cdot N_{\text{events}}} = \frac{8}{15 \cdot 8 + 3 \cdot 4} & \text{, if type}(q) == \text{SplitReal64} \\ \frac{4 \cdot N_{\text{events}}}{15 \cdot 8 \cdot N_{\text{events}} + 3 \cdot 4 \cdot N_{\text{events}}} = \frac{4}{15 \cdot 8 + 3 \cdot 4} & \text{, if type}(q) == \text{SplitInt23} \end{cases}$$

5.4.2.2 Stripping the dataset size from the maximum speedup

Assuming that we can express $T^{\text{multithread}}_{\text{decompress}}$ using $\lambda^{\text{CPU-unithread}}_{\text{decompress}}$ lets now show that max (Speedup) does not depend on the dataset size.

$$\begin{aligned}
\max(\text{Speedup}) &= \frac{\frac{S^{\text{uncompressed}}_{\text{produced}}}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{\frac{S^{\text{compressed}}_{\text{processed}}}{\lambda^{\text{H2D}}(C)} + \frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{D2H}}(C \cdot r)}} \\
&= \frac{\frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{\frac{S^{\text{compressed}}_{\text{processed}}}{\lambda^{\text{H2D}}(C)} + \frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{D2H}}(C \cdot r)}} \\
&= \frac{\frac{r}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{\frac{1}{\lambda^{\text{H2D}}(C)} + \frac{r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{r}{\lambda^{\text{D2H}}(C \cdot r)}}
\end{aligned} \tag{5.21}$$

For RDataFrame applications which satisfy the discussed assumptions, we now know that the speedup is at some point equal to expression 5.21.

5.4.2.3 Proving that the maximum speedup expression is an upper bound

We have not yet proven that for these applications, the speedup is always smaller than or equal to this expression. Essentially, we want to prove the following statement.

$$\frac{\frac{S^{\text{uncompressed}}_{\text{produced}}}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{T_{\text{D2H}} + T_{\text{decompress}} + T_{\text{H2D}}} \leq \frac{\frac{r}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{\frac{1}{\lambda^{\text{H2D}}(C)} + \frac{r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{r}{\lambda^{\text{D2H}}(C \cdot r)}}$$

With some algebraic manipulations, we can rewrite in the following way.

$$\begin{aligned}
\frac{S^{\text{uncompressed}}_{\text{produced}}}{T_{\text{D2H}} + T_{\text{decompress}} + T_{\text{H2D}}} &\leq \frac{r}{\frac{1}{\lambda^{\text{H2D}}(C)} + \frac{r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{r}{\lambda^{\text{D2H}}(C \cdot r)}} \\
S^{\text{uncompressed}}_{\text{produced}} &\leq \frac{r}{\frac{1}{\lambda^{\text{H2D}}(C)} + \frac{r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{r}{\lambda^{\text{D2H}}(C \cdot r)}} (T_{\text{D2H}} + T_{\text{decompress}} + T_{\text{H2D}})
\end{aligned}$$

$$\begin{aligned}
S^{\text{uncompressed}}_{\text{produced}} \left(\frac{1}{\lambda^{\text{H2D}}(C)} + \frac{r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{r}{\lambda^{\text{D2H}}(C \cdot r)} \right) &\leq r \cdot (T_{\text{D2H}} + T_{\text{decompress}} + T_{\text{H2D}}) \\
\frac{S^{\text{uncompressed}}_{\text{produced}}}{r} \left(\frac{1}{\lambda^{\text{H2D}}(C)} + \frac{r}{\lambda^{\text{GPU}}_{\text{decomp}}(C)} + \frac{r}{\lambda^{\text{D2H}}(C \cdot r)} \right) &\leq T_{\text{D2H}} + T_{\text{decompress}} + T_{\text{H2D}} \\
\frac{S^{\text{compressed}}_{\text{processed}}}{\lambda^{\text{H2D}}(C)} + \frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{D2H}}(C \cdot r)} &\leq T_{\text{D2H}} + T_{\text{decompress}} + T_{\text{H2D}}
\end{aligned} \tag{5.22}$$

So in order to prove the statement, it suffices to prove that $\frac{S^{\text{compressed}}_{\text{processed}}}{\lambda^{\text{H2D}}(C)} \leq T_{\text{H2D}}$, $\frac{S^{\text{compressed}}_{\text{processed}} r}{\lambda^{\text{D2H}}(C \cdot r)} \leq T_{\text{D2H}}$ and $\frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} \leq T_{\text{decompress}}$. We will proceed and do this. For this we will make use of the following statement, to which we will refer as lemma 1.

Lemma 1 If $A, B, C, D > 0$ and $D \leq C$ then $\frac{A}{C} \leq \frac{\lfloor A/B \rfloor \cdot B}{C} + \frac{A\%B}{D}$.

Proof In order to prove this statement, first note that we can write the division A/B as its fractional part added to its integer part. More formally, $\frac{A}{B} = \lfloor A/B \rfloor + \frac{A\%B}{B}$. We will now apply some algebraic manipulation to this statement.

$$\lfloor A/B \rfloor \cdot B + A\%B = A \tag{5.23}$$

$$\frac{\lfloor A/B \rfloor \cdot B}{C} + \frac{A\%B}{C} = \frac{A}{C} \tag{5.24}$$

By assumption, we know that $D \leq C$. Due to this we know that $\frac{A\%B}{D} \geq \frac{A\%B}{C}$. With this inequality and equation 5.24, we can state the desired inequality as follows.

$$\frac{\lfloor A/B \rfloor \cdot B}{C} + \frac{A\%B}{D} \geq \frac{A}{C}$$

$$\frac{A}{C} \leq \frac{\lfloor A/B \rfloor \cdot B}{C} + \frac{A\%B}{D}$$

This completes the proof of lemma 1.

Lets now start by proving that $\frac{S^{\text{compressed}}_{\text{processed}}}{\lambda^{\text{H2D}}(C)} \leq T_{\text{H2D}}$. We start from the definition of T_{H2D} which we repeat below.

$$T_{\text{H2D}} = \frac{\lfloor S^{\text{compressed}}_{\text{processed}}/C \rfloor \cdot C}{\lambda^{\text{H2D}}(C)} + \frac{S^{\text{compressed}}_{\text{processed}}\%C}{\lambda^{\text{H2D}}(S^{\text{compressed}}_{\text{processed}}\%C)}$$

Regardless of the hardware it seems likely that the host to device transfer speed never decreases, for increasing transfer buffer sizes. In other words, for all $C_1 \geq C_2$, $\lambda^{\text{H2D}}(C_1) \geq \lambda^{\text{H2D}}(C_2)$. We can also say, that for all $C_1 \leq C_2, \lambda^{\text{H2D}}(C_1) \leq \lambda^{\text{D2H}}(C_2)$. Due to the fact that regardless of C, $S^{\text{compressed}}_{\text{processed}}\%C < C$, we know that $\lambda^{\text{H2D}}(S^{\text{compressed}}_{\text{processed}}\%C) \leq \lambda^{\text{H2D}}(C)$. Obviously $S^{\text{compressed}}_{\text{processed}}, C, \lambda^{\text{H2D}}(C)$ and $\lambda^{\text{H2D}}(S^{\text{compressed}}_{\text{processed}}\%C)$ are also positive. Given this info, lemma 1 is directly applicable to the definition of T_{H2D}, yielding the statement which we wished to prove.

We will now start proving the second statement $\frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} \leq T_{\text{decompress}}$. Yet again

we take the definition of $T_{\text{decompress}}$ as a starting point.

$$\begin{aligned} T_{\text{decompress}} &= \frac{\lfloor S^{\text{compressed}}_{\text{processed}}/C \rfloor \cdot C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{S^{\text{uncompressed}}_{\text{produced}} - \lfloor S^{\text{compressed}}_{\text{processed}}/C \rfloor \cdot C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(S^{\text{compressed}}_{\text{processed}} \% C)} \\ &= \frac{\lfloor S^{\text{uncompressed}}_{\text{produced}}/(C \cdot r) \rfloor \cdot C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{S^{\text{uncompressed}}_{\text{produced}} - \lfloor S^{\text{uncompressed}}_{\text{produced}}/(C \cdot r) \rfloor \cdot C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(S^{\text{compressed}}_{\text{processed}} \% C)} \end{aligned}$$

By rewriting $S^{\text{compressed}}_{\text{processed}}$ in terms of $S^{\text{uncompressed}}_{\text{produced}}$ we can apply the identity $A - \lfloor A/B \rfloor \cdot B = A \% B$ to the numerator of the second term. One can deduce this identity from the addition of the fractional and integer part of A/B.

$$T_{\text{decompress}} = \frac{\lfloor S^{\text{uncompressed}}_{\text{produced}}/(C \cdot r) \rfloor \cdot C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{S^{\text{uncompressed}}_{\text{produced}} \%(C \cdot r)}{\lambda^{\text{GPU}}_{\text{decompress}}(S^{\text{compressed}}_{\text{processed}} \% C)}$$

It is intuitive that the GPU decompression speed does not decrease for increasing compressed buffer sizes, as it can then utilise more parallelism. Making this assumption, allows us to apply lemma 1. Rewriting $S^{\text{uncompressed}}_{\text{produced}}$ in terms of $S^{\text{compressed}}_{\text{processed}}$ then yields the statement which we wished to prove.

$$\begin{aligned} &\frac{S^{\text{uncompressed}}_{\text{produced}}}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} \leq \frac{\lfloor S^{\text{uncompressed}}_{\text{produced}}/(C \cdot r) \rfloor \cdot C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{S^{\text{uncompressed}}_{\text{produced}} \%(C \cdot r)}{\lambda^{\text{GPU}}_{\text{decompress}}(S^{\text{compressed}}_{\text{processed}} \% C)} \\ &\frac{S^{\text{compressed}}_{\text{processed}} \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}} \leq T_{\text{decompress}} \end{aligned}$$

The proof of the third statement $\frac{S^{\text{compressed}}_{\text{processed}} r}{\lambda^{\text{D2H}}(C \cdot r)} \leq T_{\text{D2H}}$ is very much alike the other two. As such, we only give a summary of this proof. One should take the definition of T_{D2H} as a starting point. After rewriting each dataset size term in terms of $S^{\text{uncompressed}}_{\text{produced}}$, it is again possible to apply lemma 1. For this we should make the intuitive assumption that the device to host transfer speed never decreases for increasing transfer sizes. Finally, writing $S^{\text{uncompressed}}_{\text{produced}}$ in terms of $S^{\text{compressed}}_{\text{processed}}$ yields the statement which we wanted.

We have now proved the three inequalities necessary to prove inequality 5.22. At this point, we thus know that equation 5.21 governs the maximum speedup of an RDataFrame program, dataset type combination whenever two conditions hold. First of all, one should be able to model the single and multi-threaded decompression time using $S^{\text{uncompressed}}_{\text{produced}}/(N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}})$. One can also restate this in two assumptions on the more granular model of $T^{\text{multithread}}_{\text{decompress}}$. Finally, the decompression speed and transfer speeds in either direction, should never decrease with increasing compressed buffer sizes.

5.4.2.4 Impact of the compression ratio on speedup predictions

Up till this point we made decompression offloading speedup predictions for the LHCB benchmark, applied on B2HHH datasets. Remember that we only consider a node with an AMD 7220P CPU running at 2.0 GHZ, and a NVIDIA A6000 GPU. By using the applicable maximum speedup equation 5.21, we will now make an analysis of the impact of the compression ratio on these speedups. Using this equation instead of the full speedup model, allows us to conveniently strip out the already known effect of the dataset size dimension.

We consider the impact of increasing compression ratio's, starting from the original compression ratio's r of all used columns. These have values 1.45, 1.41 and 1.46 for ZLIB, LZ4 and ZSTD, respectively. For the host to device transfer bandwidth and vice versa, we simplify usage of equation 5.21 by using their maximum values of 22.32 GB/sec and 10.74 GB/sec, respectively. More specifically, we replace the terms $\lambda^{\text{H2D}}(C)$ and $\lambda^{\text{D2H}}(C \cdot r)$ with these values. This does not affect the predicted speedups majorly, because we already assume that C is set so that we reach at least 95% of these maximum transfer speeds. For the same reason, we ignore the negligible increase of $\lambda^{\text{D2H}}(C \cdot r)$ for increasing values of r. Similarly, we simplify the maximum speedup prediction equation, by replacing $\lambda^{\text{GPU}}_{\text{decompress}}(C)$ with its maximum for each considered compression algorithm. These values are 85910.9 MB/sec, 73081.4 MB/sec and 11390.7 MB/sec for ZSTD, LZ4 and ZLIB, respectively.

Before making predictions of the maximum speedup in terms of r and N_{threads}, we still need $\lambda^{\text{CPU-unithread}}_{\text{decompress}}$ for each compression algorithm. We can compute the values of this parameter, for each considered compression algorithm, in terms of the known decompression speeds for each column type, compression algorithm/level combination, which table 5.2 provides. For this we use equation 5.20, as well as the known ratio's of uncompressed data within each used column with respect to the total amount of uncompressed data across all used columns, for B2HHH datasets. We determine these ratio's in section 5.4.2.1. All in all, the single-threaded decompression speeds $\lambda^{\text{CPU-unithread}}_{\text{decompress}}$ turn out as 0.34 GB/sec, 5.2 GB/sec and 2.3 GB/sec for ZLIB, LZ4 and ZSTD, respectively.

Figure 5.14 shows that increasing compression ratio's slightly improve the maximum GPU decompression offloading speedup with respect to CPU decompression, across all thread counts, for datasets compressed using one of the considered compression algorithms. For the highest compression ratio we get the highest predicted speedup. On our hardware, we have some certainty that the green speedup predictions in figure 5.14 are true for big

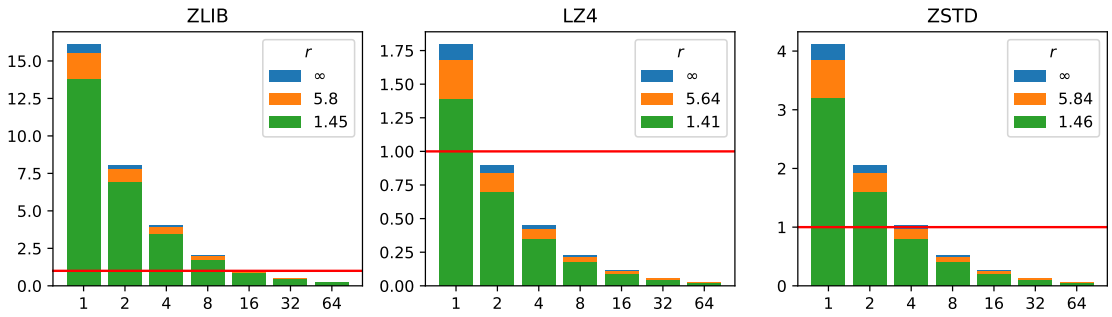


Figure 5.14: Shows the predicted maximum speedup of GPU decompression offloading with respect to CPU decompression, across various thread counts, for each of the three considered compression algorithms, when varying the compression ratio. These maximum speedups assume the absence of a remainder compressed buffer that inherently suffers from lower decompression and transfer speeds. We also assume usage of a node with an AMD EPYC 7220P @ 2.0 GHZ and a NVIDIA A6000 GPU. In each graph, the green bars display the predicted speedups for the original compression ratio r, the orange bars for a r twice as big, and finally the blue bars display predictions for an infinitely big r.

enough B2HHH datasets, because we measured the CPU decompression speeds, as well as the GPU decompression speeds, for datasets of this type, with these r values. Predictions for increasing r values rely on the assumption that the CPU and GPU decompression speeds do not change as a result.

One might wonder what datasets belong to the predictions for higher compression ratio's. In order to answer this question, lets revisit our definition of a dataset type. We define a dataset type as a set of fields, with certain names and types, as well as certain field value distributions. Throughout this thesis, we assume that RDataFrame programs are meant to operate on certain datasets, with a specific type. For the LHCB benchmark we name this type after its corresponding B2HHH dataset, the B2HHH type. As mentioned, we know that the green speedup predictions belong to datasets of the B2HHH type, in the context of our hardware and the LHCB benchmark. Datasets with higher r values must then contain the same field names, and types not too different, for else the LHCB benchmark can not act on them. More specifically, a field type such as an integer can for example be replaced with a float, but not with a vector. Different field value distributions and/or types must thus drive the higher r values. Because of these differences, the higher r values thus correspond to datasets with a type other than the B2HHH type.

When evaluating the following slightly rewritten variant of equation 5.21, we can see that for all RDataFrame application, dataset combinations for which this equation holds,

increasing r values improve the speedup through reducing the term related to T_{H2D}.

$$\max\left(\text{Speedup}\right) = \frac{\frac{1}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{\frac{1}{r\lambda^{\text{H2D}}(C)} + \frac{1}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{1}{\lambda^{\text{D2H}}(C \cdot r)}}$$

In order to gain some more intuition as to why this is the case, we introduce the following three metrics, that describe the time fraction of each part of the GPU decompression implementation, with respect to its total latency. We compute these fractions for the maximum speedup predictions, in other words for dataset sizes whom are a multiple of C. Therefore we can write each T term in the simple way presented here.

$$\frac{T_{\text{D2H}}}{T_{\text{D2H}} + T_{\text{decompress}} + T_{\text{H2D}}} = \frac{\frac{1}{\lambda^{\text{D2H}}(C \cdot r)}}{\frac{1}{r\lambda^{\text{H2D}}(C)} + \frac{1}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{1}{\lambda^{\text{D2H}}(C \cdot r)}} \tag{5.25}$$

$$\frac{T_{\text{decompress}}}{T_{\text{D2H}} + T_{\text{decompress}} + T_{\text{H2D}}} = \frac{\frac{1}{\lambda^{\text{GPU}}_{\text{decompress}}(C)}}{\frac{1}{r\lambda^{\text{H2D}}(C)} + \frac{1}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{1}{\lambda^{\text{D2H}}(C \cdot r)}} \tag{5.26}$$

$$\frac{T_{\text{H2D}}}{T_{\text{D2H}} + T_{\text{decompress}} + T_{\text{H2D}}} = \frac{\frac{1}{\lambda^{\text{H2D}}(C)}}{\frac{1}{\lambda^{\text{H2D}}(C)} + \frac{r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{r}{\lambda^{\text{D2H}}(C \cdot r)}} \tag{5.27}$$

Lets now evaluate these fractions for increasing r values. For the fraction of T_{D2H} with respect to the total GPU decompression offloading latency, we see that increasing r values decrease the term $1/(r\lambda^{\text{H2D}}(C))$, which means that the fraction as a whole increases. We can make the same observations for the fraction of $T_{\text{decompress}}$. On the contrary, for the fraction of T_{H2D}, increasing r values enlarge the terms $r/(\lambda^{\text{GPU}}_{\text{decompress}}(C))$ and $r/(\lambda^{\text{D2H}}(C \cdot r))$ which decreases the value of the fraction as a whole. Starting from a RDataFrame application dataset combination, where $\max\left(\text{Speedup}\right)$ holds, increasing the compression ratio thus improves the predicted speedup and reduces the fraction of time spent by the GPU decompression offloading implementation, on transferring data to the GPU, while simultaneously enlarging the fraction of time spent on GPU decompression and device to host data transfers. We say starting from a RDataFrame application dataset combination, because increasing the compression ratio changes the dataset type.

We can get the following expression when taking the limit of r to infinity, where the term related to T_{D2H} drops to zero. If $\lambda^{\text{D2H}}(x)$ increases to a horizontal asymptote, which is likely, then one can interpret $\lim_{r\to\infty} \lambda^{\text{D2H}}(C \cdot r)$ as the maximum D2H transfer speed. For our hardware, we found out that this is the case.

$$\lim_{r\to\infty} \max\left(\text{Speedup}\right) = \frac{\frac{1}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{\frac{1}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{1}{\lim_{r\to\infty} \lambda^{\text{D2H}}(C \cdot r)}}$$

For a RDataFrame program where max (Speedup) holds, that uses a compressed dataset of a certain type, as well as specific hardware, we can determine an expression for the compression ratio $r_{95\%}$ for which we reach 95% of $\lim_{r\to\infty} \max(\text{Speedup})$, by solving the following equation for this variable. Note that this $r_{95\%}$ belongs to a different RDataFrame program dataset type combination, because changing r changes the dataset type.

$$\frac{\frac{1}{N_{\text{threads}}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}{\frac{1}{r_{95\%} \cdot \lambda^{\text{H2D}}(C)} + \frac{1}{\lambda^{\text{GPU}}_{\text{decompress}}} + \frac{1}{\lim_{r\to\infty} \lambda^{\text{D2H}}(C \cdot r)}} = 0.95 \cdot \frac{\frac{1}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{\frac{1}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{1}{\lim_{r\to\infty} \lambda^{\text{D2H}}(C \cdot r)}}$$

Solving the equation shown above, yields the following expression, that does not depend on the amount of threads.

$$r_{95\%} = \frac{19 \cdot \lambda^{\text{GPU}}_{\text{decompress}}(C) \cdot \lim_{r\to\infty} \left[\lambda^{\text{D2H}}(C \cdot r)\right]}{\lambda^{\text{H2D}}(C) \cdot \left(\lambda^{\text{GPU}}_{\text{decompress}}(C) + \lim_{r\to\infty} \left[\lambda^{\text{D2H}}(C \cdot r)\right]\right)}$$

Using this equation, the λ functions for our hardware, and our values of C, we get a value $r_{95\%}$ of 4.71, 7.97 and 8.13 for ZLIB, LZ4 and ZSTD, respectively.

5.4.2.5 Scenario analysis

A powerful use-case of the maximum speedup prediction model, is analysing the impact of various scenarios on the maximum speedups. In this section, we will showcase the impact of scenarios such as introducing a faster method to perform D2H and H2D transfers, overlapping communication and computation, as well as the impact of PCIE 5.0, which is an upcoming transfer link standard. We will discuss scenarios along a path for each considered compression algorithm, in the primary order of current applicability, and a secondary order of performance improving effectivity.

Note that the validity of this analysis will be constrained to our hardware, and the LHCB benchmark applied on datasets with the B2HHH type or a closely related dataset type. With a closely related type we specifically refer to datasets with the same field names as in B2HHH datasets, where a difference in field types and/or field value distributions, enables an increase in compression ratio's. These constraints follow from the calibration context of max (Speedup). For this analysis to hold, we additionally need to assume that increasing r-values do not change the CPU and GPU decompression speeds.

For each of the considered compression algorithms, Figure 5.14 displays the predicted speedup of GPU decompression offloading, for various compression ratios. In order to choose next scenarios that might improve this, lets look at the corresponding GPU decompression offloading time distributions in Figure 5.15. Each column of graphs in this figure

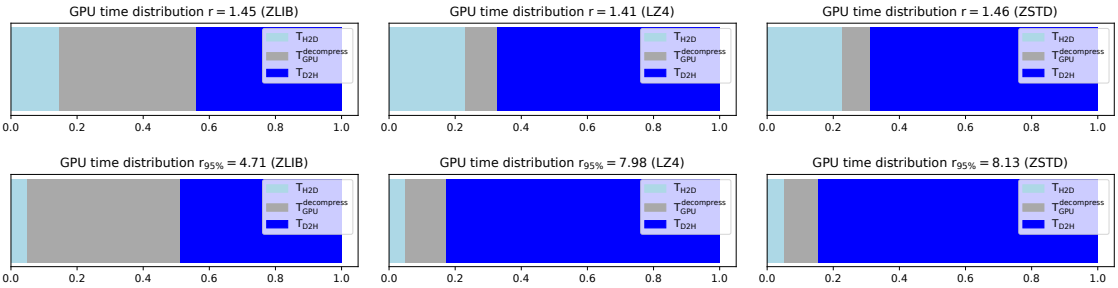


Figure 5.15: The GPU decompression offloading time distributions, for B2HHH datasets and closely related dataset types with a higher compression ratio , in the context of our hardware. Each column displays these time distributions for a specific compression algorithm, where the top, and bottom graph show distributions for B2HHH datasets and closely related dataset types with a higher compression ratio, respectively. In each distribution light blue, grey and dark blue indicate the H2D transfer, GPU decompression and D2H transfer time fractions, respectively.

describes these distributions when using a specific compression algorithm. The top graph of each column displays this distribution for B2HHH datasets, whereas the bottom graph displays this distribution for a closely related dataset type, where we increase the compression ratio to $r_{95\%}$. Remember that for $r_{95\%}$ we reach 95% of the maximum speedup, as a result of increasing the compression ratio. The light blue, grey and dark blue color in each distribution show the time fraction spent on H2D transfers, GPU decompression and D2H transfers, respectively. For each compression algorithm, we compute these time distributions using equations 5.25, 5.26 and 5.27, where we use the same values for $\lambda^{\text{H2D}}(C)$, $\lambda^{\text{GPU}}_{\text{decompress}}(C)$ and $\lambda^{\text{D2H}}(C \cdot r)$ as in the previous section.

We can see that for ZLIB, neither data transfers or computation are a convincing GPU decompression offloading performance bottleneck. On the other hand, for LZ4 and ZSTD, data transfers are a clear performance bottleneck. Regardless of the compression algorithm, data transfers still occupy quite some time. As such, we can expect that reducing this time, will improve the GPU decompression offloading speedup across all thread counts, for all considered compression algorithms.

Pinned memory One approach to achieve this, is to use pinned memory in the host. More specifically, we should build buffers of compressed pages, in pinned memory, and transfer data from this type of memory to the GPU. Additionally, we should receive the decompressed buffer counterparts, in pinned memory as well. Another name for pinned memory is page-locked memory. As implied by its name, data allocated in this type of

memory, can not be paged out to secondary storage, during program execution.

In order to understand why the usage of pinned memory allows higher host-device transfer bandwidths in either direction, we need to go into the operation of such transfers (92, p. 464). For NVIDIA GPUs, a regular data transfer from the host to the device, starts by copying the to-be transferred buffer, into a page-locked memory buffer. Then, the GPU fetches the data from main memory in its memory, through a *Direct Memory Access* (DMA) request on the physical memory regions corresponding to this page-locked memory buffer. If we were to skip the step of copying the to-be transferred buffer in pinned memory, pages could be paged out to secondary storage and replaced by others, at any point during the data transfer, thus corrupting it.

For a regular device to host memory transfer, a DMA transfers data from the device to the host in a page-locked memory buffer. Only afterwards, the data is copied into the destination pageable memory buffer. Yet again, we can not copy directly into pageable memory, because the physical memory regions corresponding to these pages, can change in terms of the pages that they store, during the transfer. Now one can see that for both transfer directions, direct usage of pinned memory prevents the need to copy data into such a buffer first, which improves the transfer bandwidths.

For us to predict speedups with max (Speedup) when using pinned memory, for transferring compressed buffers and receiving their decompressed counterparts, we need to find the corresponding transfer bandwidths. For this, we slightly modified the benchmark used to find the transfer bandwidths for pageable host memory, described in section 5.1.5.2. Instead of using `malloc` to allocate host memory, we now simply use `cudaAllocHost`. Using this slightly modified benchmark, we found a H2D and D2H transfer bandwidth of 25.03 GB/sec and 26.19 GB/sec for big pinned buffer sizes, on our DAS6 node. This improves the bandwidths of transferring buffers to and from pageable memory, which had values of 22.32 GB/sec for H2D transfers, and 10.74 GB/sec for D2H transfers. We can see that usage of pinned memory, especially improves the D2H transfer bandwidth.

Now that we know the values of $\lambda^{\text{H2D}}(C)$ and $\lambda^{\text{D2H}}(C \cdot r)$ when using pinned memory, we can compute the predicted max (Speedup) of GPU decompression offloading when using that memory, for B2HHH datasets when using ZLIB, LZ4 or ZSTD. In this computation we also need the CPU/GPU decompression speeds for each algorithm, shown in the previous section. After recomputing $r_{95\%}$ due to changing H2D and D2H transfer speeds, we can also make these predictions for the related datasets with $r_{95\%}$. We show the speedup predictions for r and $r_{95\%}$ in the top row of graphs, in Figure 5.16. Each column of graphs corresponds

to a compression algorithm, and the distributions below each speedup prediction show the GPU decompression offloading time distribution for r and $r_{95\%}$.

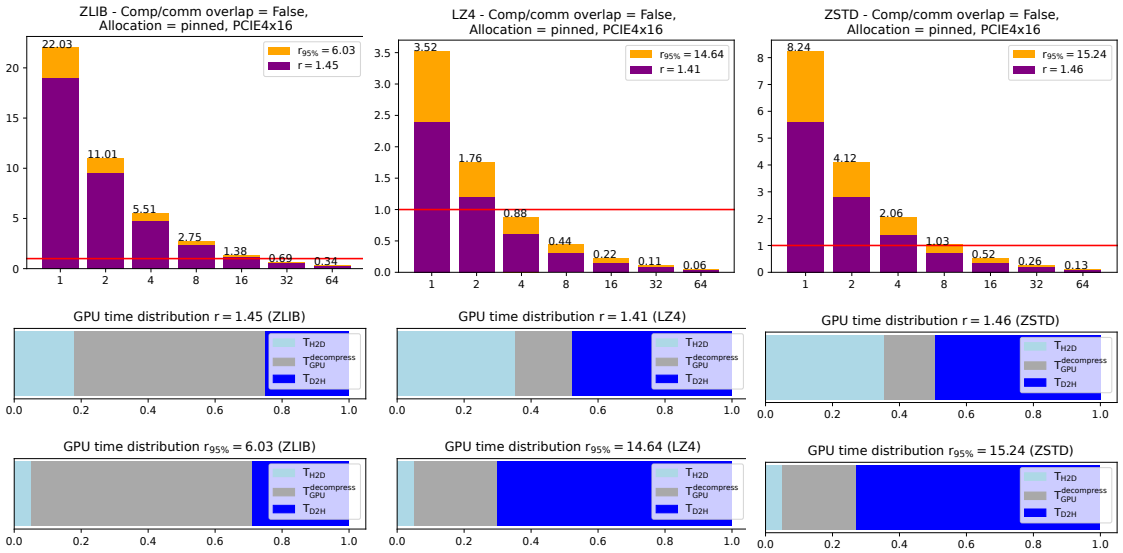


Figure 5.16: In the top row of graphs, this figure displays the predicted speedups of GPU decompression offloading with respect to CPU decompression on our hardware across various thread counts, for B2HHH datasets (purple bars), as well as closely related dataset types with a higher compression ratio $r_{95\%}$ (orange bars), when applying ZLIB, LZ4 or ZSTD compression. This GPU decompression offloading implementation uses pinned memory in the host, for transferring compressed buffers and their decompressed counterparts. Each column, displaying graphs for a specific compression algorithm, also contains the GPU decompression offloading time distribution, for r and $r_{95\%}$, where the light blue, grey and dark blue colors refer to the H2D transfer time, the GPU decompression time, and the D2H transfer time, respectively.

Comparing the speedup predictions in Figure 5.16, with those made in Figure 5.14 where we assumed usage of pageable memory, reveals that they are higher in Figure 5.16 across all CPU thread counts and algorithms. The performance benefit of using pinned memory is in line with our expectation, as well as the cause, which is a smaller amount of time spent on transferring data. One can easily observe this cause, when comparing the GPU decompression time distributions in Figure 5.16 with those made in Figure 5.15 for the usage of pageable memory.

We can also describe the performance benefit of using pinned memory, in terms of the amount of CPU threads that GPU decompression offloading can beat, with respect to these numbers, when using pageable memory. With pageable memory, GPU decompression offloading can beat at most 13 (r) or 15 CPU threads ($r_{95\%}$) for ZLIB. For LZ4 and ZSTD,

GPU decompression offloading can beat at most 1 and 3 CPU threads, respectively. When using pinned memory we improve these numbers to (21, 18), (3,2) and (8,5), for ZLIB, LZ4 and ZSTD, respectively. In these tuples the first and second number refer to the amount of CPU threads that GPU decompression offloading can beat, for $r_{95\%}$ and r, respectively.

Looking at the GPU decompression offloading time distributions in Figure 5.16 we can see that transfer times are still a clear bottleneck for LZ4 and ZSTD. On the other hand, for ZLIB, the GPU decompression time is now the bottleneck. As a next scenario we will look at an approach that overlaps H2D transfers, GPU decompression and D2H transfers. Ideally, this would enable the three blocks of each GPU decompression offloading time distribution in Figure 5.16, to overlap in execution, which would clearly improve speedups across all considered algorithms.

Overlapping data transfers and GPU decompression In this scenario, we will evaluate the effect of additionally introducing an overlap of H2D transfers, D2H transfers, and GPU decompression, on the speedup of GPU decompression offloading. In order to understand this idea, lets first look at the top diagram in Figure 5.17. This illustrates the execution of an RDataFrame program over time, that uses buffered GPU decompression offloading, without such an overlap. In this diagram, the light blue, grey and dark blue blocks refer to the time spent on D2H transfers, GPU decompression and H2D transfers, respectively. The green blocks refer to the time spent on all other operations necessary for the execution of RDataFrame programs.

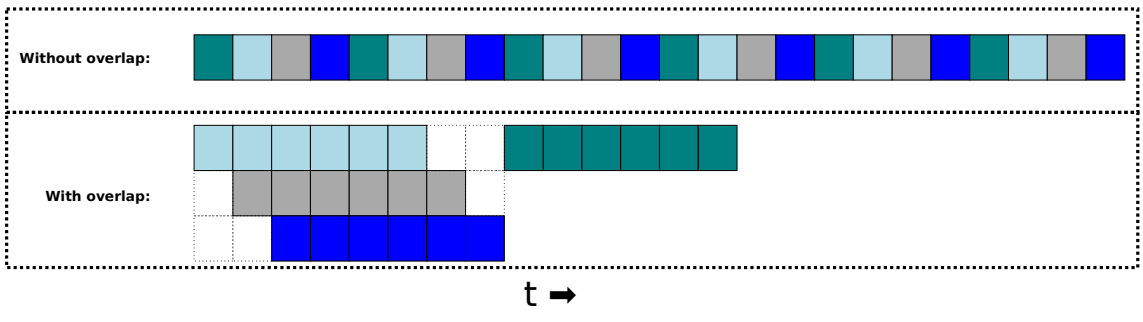


Figure 5.17: The top diagram displays the performed operations in the execution of RDataFrame programs over time that use buffered GPU decompression offloading. In the bottom diagram, we display the execution of such programs over time where we allow an overlap of H2D transfers, GPU decompressions and D2H transfers. In both diagrams, light blue, grey and dark blue blocks stand for time spent on H2D transfers, GPU decompression and D2H transfers. Green blocks refer to time spent on all other operations required in the execution of RDataFrame programs.

By performing the required operations for GPU decompression of many compressed buffers consecutively, it is possible to get an overlap of these operations, across different compressed buffers. For this we should use the asynchronous memory copy methods from CUDA, post the required operations for each compressed buffer to a separate stream, and additionally make use of pinned memory (93). In order to estimate the latency of decompression in this manner, we should assume a certain execution of these operations.

We assume that execution occurs according to the bottom diagram of Figure 5.17, which also occurs as an example in (93). As shown in this figure, this execution style allows one unidirectional data transfer at a time, and one kernel execution at a time. This does allow an overlap of at most one D2H transfer, one H2D transfer and one GPU decompression, each belonging to different compressed buffers. We think overlapping at most one H2D and D2H transfer is optimal for us, because we already use 80% of the available unidirectional PCIE4.0x16 bandwidth when transferring pinned buffers. Doing transfers in two directions at once, is beneficial because it allows us to utilise the 32 GB/sec bandwidth offered by PCIEe4.0x16, in each direction simultaneously (83) . Disregarding data dependencies, we do not think that decompressing multiple buffers at once, would enable a performance benefit, because we already saturate the device, by setting the compressed buffer sizes to large enough values (see Figure 5.11 and Figure 5.9).

GPU decompression offloading, as executed in the bottom diagram of Figure 5.17, seems to originate from a three stage pipeline of a H2D transfer, GPU decompression and a D2H transfer. This pipeline processes each required compressed buffer, sequentially. We can describe the total latency $T^{\text{GPU}}_{\text{decompress-pipelined}}$ of processing N_{buffers} in such a pipeline, with each stage taking up latencies $T^{\text{buffer}}_{\text{D2H}}$, $T^{\text{buffer}}_{\text{GPU-decompress}}$, and $T^{\text{buffer}}_{\text{H2D}}$, using the following equation (94).

$$\begin{aligned} T^{\text{GPU}}_{\text{decompress-pipelined}} =& T^{\text{buffer}}_{\text{H2D}} + T^{\text{buffer}}_{\text{GPU-decompress}} + T^{\text{buffer}}_{\text{D2H}} + \\ & \max\left(T^{\text{buffer}}_{\text{H2D}}, T^{\text{buffer}}_{\text{GPU-decompress}}, T^{\text{buffer}}_{\text{D2H}}\right) \cdot (N_{\text{buffers}} - 1) \end{aligned} \tag{5.28}$$

In bottom diagram of Figure 5.17 $T^{\text{buffer}}_{\text{D2H}} = T^{\text{buffer}}_{\text{GPU-decompress}} = T^{\text{buffer}}_{\text{H2D}} = 1$, and we process 6 compressed buffers. Substituting these values in equation 5.28 indeed yields the correct time value 8.

If we want to use equation 5.28 to characterise the latency of pipelined GPU decompression offloading, we implicitly assume that the H2D transfer time, and the GPU decompression time for each compressed buffer, are the same. Additionally, we assume that the D2H transfer times of each decompressed counterpart, are the same. Using this formulation, we can thus only consider modelling the pipelined GPU decompression offloading

latency, for compressed dataset sizes that are a multiple of the chunk size C. In this case, $T^{\text{buffer}}_{\text{D2H}}$, $T^{\text{buffer}}_{\text{GPU-decompress}}$ and $T^{\text{buffer}}_{\text{H2D}}$ are namely the same for each compressed buffer and its decompressed counterparts. Lets now reformulate each T term in terms of the size of the compressed/uncompressed buffer that should be produced or transferred, divided by a λ function that characterises a transfer speed or the speed at which the GPU can produce uncompressed data. We also rewrite N_{buffers} as the quotient of the amount of processed compressed data and C.

$$T^{\text{GPU}}_{\text{decompress-pipelined}} = \frac{C}{\lambda^{\text{H2D}}(C)} + \frac{C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{C \cdot r}{\lambda^{\text{D2H}}(C \cdot r)} + \max\left(\frac{C}{\lambda^{\text{H2D}}(C)}, \frac{C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)}, \frac{C \cdot r}{\lambda^{\text{D2H}}(C \cdot r)}\right) \cdot \left(\frac{S^{\text{compressed}}_{\text{processed}}}{C} - 1\right) \tag{5.29}$$

Using this equation, we can formulate the speedup of pipelined GPU decompression offloading, with respect to single or multi-threaded decompression, in the context of specific hardware, an RDataFrame program and a corresponding dataset type, when using a certain compression algorithm. As described previously, we are also restricted to making predictions for compressed dataset sizes that are a multiple of C. Because we want to make this speedup independent of the dataset size, we also require that one can express the single and multi-threaded decompression latency in terms of a single decompression speed. We discussed this condition in terms of the original model that uses separate decompression speeds per column, in section 5.4.2.1.

We will now formulate Speedup(Pipelined) in this context. Starting from the quotient defining this speedup, we rewrite each model in terms of its definition. Then we perform some algebra to yield an intermediate result. In these equations we use M to temporarily denote $\max\left(\frac{C}{\lambda^{\text{H2D}}(C)}, \frac{C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)}, \frac{C \cdot r}{\lambda^{\text{D2H}}(C \cdot r)}\right)$.

$$\begin{aligned}
&\text{Speedup(Pipelined)} = T^{\text{multithread}}_{\text{decompress}} / T^{\text{GPU}}_{\text{decompress-pipelined}} \\
&= \frac{\frac{S^{\text{uncompressed}}_{\text{produced}}}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{\frac{C}{\lambda^{\text{H2D}}(C)} + \frac{C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{C \cdot r}{\lambda^{\text{D2H}}(C \cdot r)} + M \cdot \left(\frac{S^{\text{compressed}}_{\text{processed}}}{C} - 1\right)} \\
&= \frac{\frac{S^{\text{uncompressed}}_{\text{produced}}}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{\frac{C}{\lambda^{\text{H2D}}(C)} + \frac{C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{C \cdot r}{\lambda^{\text{D2H}}(C \cdot r)} - M + M \cdot \frac{S^{\text{uncompressed}}_{\text{produced}}}{r \cdot C}} \\
&= \frac{\frac{1}{N_{\text{threads}} \cdot \lambda^{\text{CPU-unithread}}_{\text{decompress}}}}{\frac{1}{S^{\text{uncompressed}}_{\text{produced}}}\left(\frac{C}{\lambda^{\text{H2D}}(C)} + \frac{C \cdot r}{\lambda^{\text{GPU}}_{\text{decompress}}(C)} + \frac{C \cdot r}{\lambda^{\text{D2H}}(C \cdot r)} - M\right) + M \cdot \frac{1}{r \cdot C}}
\end{aligned} \tag{5.30}$$

The final equation 5.30 reveals a dependency on the amount of produced uncompressed data. More specifically, increasing the value of $S_{\text{produced}}^{\text{uncompressed}}$, will lower the value of the first term in the denumerator. As a result, the predicted pipelined GPU decompression offloading speedup, will increase. In other words, given a certain dataset type, hardware and chunk size setting, we reach an optimum speedup for very big dataset sizes. This speedup is bounded by the following equation, obtained by taking the limit of Speedup(Pipelined) with respect to $S_{\text{produced}}^{\text{uncompressed}}$.

$$\begin{aligned}\lim_{S_{\text{produced}}^{\text{uncompressed}} \to \infty} [\text{Speedup(Pipelined)}] &= \frac{\frac{1}{N_{\text{threads}} \cdot \lambda_{\text{decompress}}^{\text{CPU-unithread}}}}{\frac{1}{r \cdot C} \max\left(\frac{C}{\lambda^{\text{H2D}}(C)}, \frac{C \cdot r}{\lambda_{\text{decompress}}^{\text{GPU}}(C)}, \frac{C \cdot r}{\lambda^{\text{D2H}}(C \cdot r)}\right)} \\ &= \frac{\frac{1}{N_{\text{threads}} \cdot \lambda_{\text{decompress}}^{\text{CPU-unithread}}}}{\max\left(\frac{1}{r \cdot \lambda^{\text{H2D}}(C)}, \frac{1}{\lambda_{\text{decompress}}^{\text{GPU}}(C)}, \frac{1}{\lambda^{\text{D2H}}(C \cdot r)}\right)} \end{aligned} \tag{5.31}$$

Given equation 5.31, we will make predictions of the pipelined GPU decompression offloading speedup, for big dataset sizes, in the context of our hardware, and the LHCB benchmark applied on B2HHH datasets. As before, we make these predictions for each considered algorithm, in terms of the amount of CPU threads.

For this, we need the B2HHH dataset r-values, GPU decompression speeds, and CPU decompression speeds, provided by section 5.4.2.4. In addition we need the H2D and D2H transfer speeds with pinned memory, whom can be found in section 5.4.2.5. To make predictions, we replace all corresponding terms in equation 5.31 with these values. Note that we assume that C is always chosen to reach most of the available transfer and GPU decompression speeds. Additionally, we assume that bidirectional data transfers do not have slower transfer speeds, then those values found for unidirectional transfers.

We present the predicted speedups of pipelined GPU decompression offloading in the top row of Figure 5.18. These speedup predictions, outperform all speedup predictions shown in Figure 5.16, for buffered GPU decompression offloading with pinned memory but wihout pipelining. In other words, pipelined GPU decompression offloading, can now beat up till 32, 4 and 11 CPU threads, for ZLIB, LZ4 and ZSTD, respectively. GPU decompression offloading without pipelining, could only beat (21, 18), (3, 2) and (8,5) CPU threads, for ZLIB, LZ4, and ZSTD, respectively. Remember that each tuple indicates a speedup for $(r, r_{95\%})$.

There is a "GPU time distribution graph" beneath each speedup prediction graph, that visualises the magnitude of each component in $\max\left(\frac{1}{r \cdot \lambda^{\text{H2D}}(C)}, \frac{1}{\lambda_{\text{decompress}}^{\text{GPU}}(C)}, \frac{1}{\lambda^{\text{D2H}}(C \cdot r)}\right)$, by

overlapping them. We normalised these magnitudes by their biggest value, in the context of a specific compression algorithm. Pipelined GPU decompression offloading performance is essentially bounded by the slowest operation for each compressed buffer. As can be seen in the time distributions of Figure 5.18, this is the GPU decompression time for ZLIB. For LZ4 and ZSTD, this are D2H transfers. Further performance optimisations for either algorithms, should thus focus on optimising these elements. Whereas before we investigated the effect of increasing compression ratio's, we do not do so now. That is because equation 5.31 shows, that increasing compression ratio's can only improve the pipelined GPU decompression offloading speedup, if H2D transfers of compressed buffers, dominate their decompression or D2H transfers. Because this is not the case for our considered algorithms, we do not need to explore the effect of higher compression ratio's.

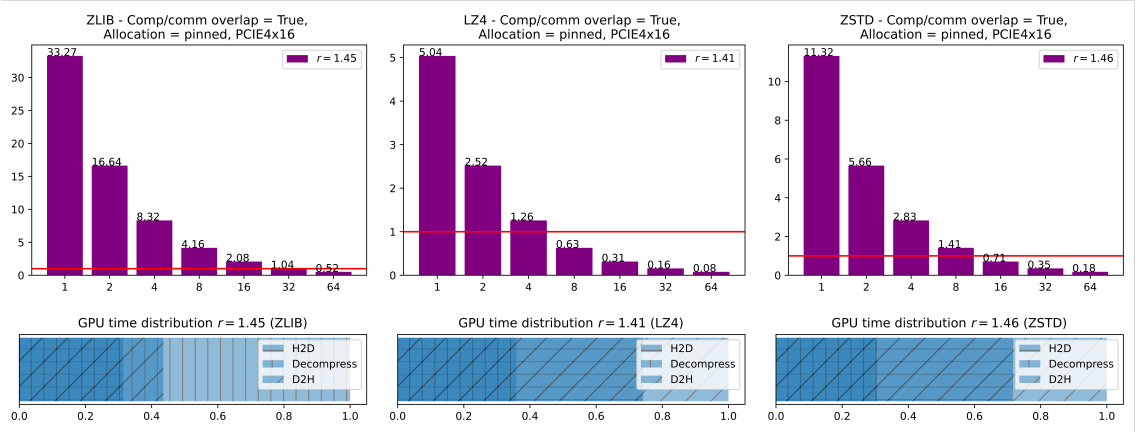


Figure 5.18: The top row of graphs display the predicted pipelined GPU decompression offloading speedup when using pinned memory on our hardware, for the LHCB benchmark and its B2HHH datasets, for each of the considered compression algorithms, across various thread counts. Beneath each speedup graph we overlay the specific argument values of $\max\left(\frac{1}{r \cdot \lambda^{\text{H2D}}(C)}, \frac{1}{\lambda^{\text{GPU}}_{\text{decompress}}(C)}, \frac{1}{\lambda^{\text{D2H}}(C \cdot r)}\right)$, normalised to their biggest value, in the context of a specific compression algorithm.

PCIE5.0 We have just observed in Figure 5.18, that the predicted speedups of pipelined GPU decompression offloading, are bounded by D2H transfers for LZ4 and ZSTD. When PCIE5 arrives, the successor of PCIE4, transfer times can be reduced through an improved maximum unidirectional bandwidth of 64 GB/sec, instead of 32 GB/sec. We will now quantify the exact speedup improvement for these algorithms, in a scenario where this new link technology is available.

For making these predictions for pipelined GPU decompression offloading, in the context

of our hardware, and the LHCB benchmark applied on B2HHH datasets, we can simply reuse equation 5.31 and the values presented in the previous section. We should only replace $\lambda^{\text{H2D}}(C)$ and $\lambda^{\text{D2H}}(C \cdot r)$ with the values for PCIE5. Previously, we saw that pinned memory transfers with PCIE4x16 yielded a maximum H2D bandwidth of 25.03 GB/sec, and a maximum D2H bandwidth of 26.19 GB/sec. For H2D and D2H, this is 78% and 82% with respect to the theoretical maximum of 32 GB/sec, respectively. Assuming that we reach these exact percentages of the maximum PCIE5x16 bandwidth, we get a value of 49.92 GB/sec for H2D transfers, and 52.48 GB/sec for D2H transfers.

Now that we have all the required values to make predictions for LZ4 and ZSTD, we present these in Figure 5.19. One can see that this indeed improves the overall speedups for LZ4 and ZSTD, with respect to those without PCIE5, that are visualised in Figure 5.18. In other words, with usage of PCIE5, we can beat at most 9 CPU threads for LZ4, and 21 CPU threads for ZSTD. Without PCIE5, these numbers were limited to 4 CPU threads for LZ4 and 11 CPU threads for ZSTD. Looking at the time distributions beneath each speedup prediction diagram, we can see that D2H transfer times are still the current performance bottleneck, however.

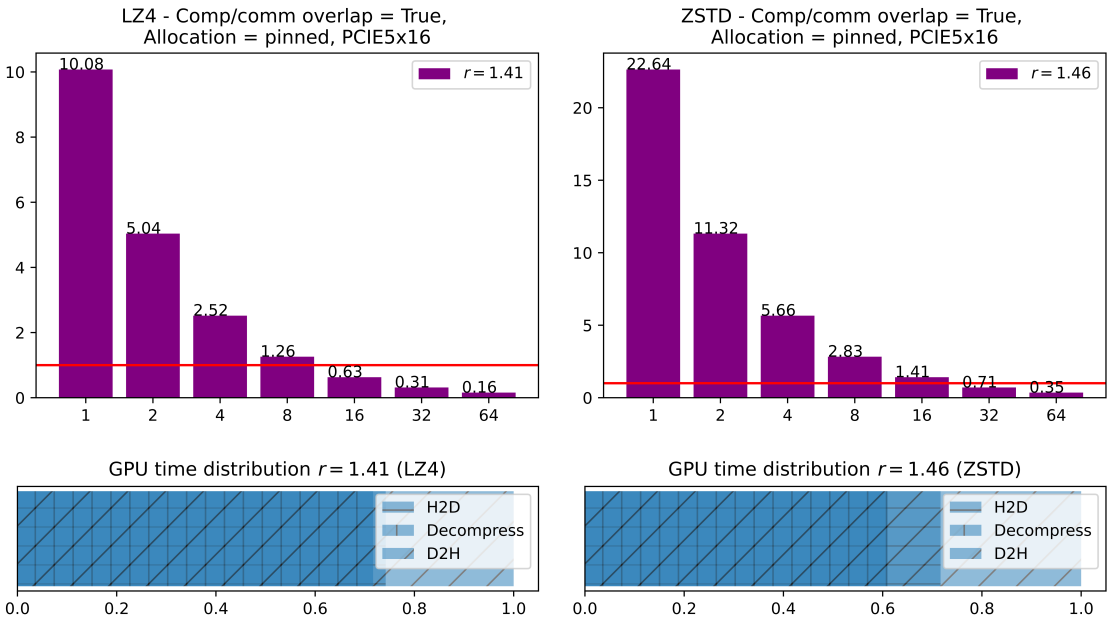


Figure 5.19: The top row of graphs display the predicted speedups of pipelined GPU decompression offloading with respect to CPU decompression, in the context of our hardware, the LHCB benchmark applied on B2HHH datasets, and PCIE5.0x16. Beneath each speedup graph we display a distribution of time, in the GPU decompression offloading implementation.

6

Conclusion

6.1 Summary and main findings

Our journey to investigate the optimisation of decompression in RDataFrame applications, started in chapter 3, with us creating two initial latency models in terms of the dataset size N_{events}, that describe the application latency. In actuality, we investigated optimising a scope bigger than decompression alone, which also includes unpacking. RNTuple datasets are packed before applying compression, in order to increase the efficiency of compression. Examples of packing include converting Boolean's in a vector of bits, and converting a list of increasing integers into their differences (delta encoding).

We created both initial models through talking with CERN, studying the documentation of the RDataFarme framework, and its code. In the first model, we model the total latency by assuming that we enter each execution path with a fixed ratio of N_{events}, and that each execution path has a fixed latency. We could not calibrate this model for the LHCB benchmark and the B2HHH dataset, by measuring the global event loop iteration latency distribution. For our second model, we therefore introduce a more fine grained calibration procedure. The model itself is also more fine grained, by separating the latency of each execution path in a data fetching and computational component. Apart from this, the model also omits some components which are unrelated to *unsealing* (unpacking & decompression). In this model, we capture both latency components of each execution path using a mean value. It turns out that this characterisation is not accurate for data fetching, because the IO system servicing these requests, provides data with a high latency variation.

In order to model this data fetching more accurately, we investigated the operation of the IO system, and subsequently incorporated this knowledge in a more detailed model, in

chapter 4. Yet again, this model omits components unrelated to unsealing. With respect to the previous model, we replace the data fetching latency component per execution path, by the sum of the page unsealing time, and the page deserialization time. We also model the time for a background thread that performs the reading of clusters of data, from disk or any other source. For us to compare the current unsealing implementation to a GPU offloading approach, we only calibrated the unsealing latency model from the bigger model, because the other parts of this bigger model are not relevant in this comparison. For the LHCB benchmark, our hardware, and datasets of the B2HHH type, this calibrated model turns out as very accurate. From now on we will refer to this context, when referring to our context. In our context, a slight modification of this model for the multi-threaded operation of RDataFrame applications, also provides great accuracy across various thread counts, when applied on the decompression component of unsealing. As such, we reverted our scope from offloading unsealing to the GPU, to the GPU offloading of decompression. Note that in this more restricted scope, the single-threaded latency model turns out to retain its previously described accuracy.

Now that we have calibrated models for the decompression time on the CPU, in a single-threaded and multi-threaded context, we need a GPU decompression offloading implementation idea, and a corresponding model, to allow us to quantify its relative performance. We start by creating both elements in chapter 5, by separating the total latency of such an implementation in transfer times to and from the GPU, as well as a GPU decompression time. Subsequently, we evaluated a data transfer approach that transfers individual pages, and buffers of pages. Transferring buffers of pages can occur faster than individual pages, because larger transfer sizes have higher transfer speeds. Because of this performance advantage, we chose to use this approach in our hypothetical GPU decompression offloading implementation. For buffered page transfers we model the transfer time with a speed for all full buffers, and speeds for remainder buffers. Likewise, we model the GPU decompression time for a compression algorithm, with a speed for decompressing full buffers, and speeds for remainder buffers. In our context, we found the required relationships between the page buffer size and the transfer / decompression speed, for each considered algorithm. For the decompression speed, we benchmarked the GPU decompression library `nvcomp` by NVIDIA.

At this point, we have a calibrated model for the CPU decompression latency in our context, for each considered compression algorithm. For such a CPU decompression latency model we can vary the dataset size and the amount of used threads. For each compression algorithm, we also have a calibrated GPU decompression offloading model, which is a

function of the dataset size. By computing the speedup as the ratio of both models, we could predict, that in our context and for big enough dataset sizes, offloading decompression outperforms the current approach, up till 13 threads for ZLIB, up till 3 threads for ZSTD, and only for a single thread for LZ4. From this data, we can also observe that for big dataset sizes, or for dataset sizes which are a multiple of the compressed buffer size, we reach the maximum speedup, irrespective of the considered algorithm.

We created an expression for this upper bound, and proved that it holds for RDataFrame programs when (1) one can model the CPU decompression latency with a single decompression speed and (2) when the transfer and decompression speeds do not decrease for bigger compressed buffer sizes. Using this expression, we determined that increasing compression ratio's improve the predicted speedups in our context. We found that a decrease in the fraction of time spent on transferring compressed data, correlates with this improved speedup. By manipulating parameters in this maximum speedup model, as well as making slight changes to its expression, we also analysed the effect of various scenarios in our context, such as using pinned memory in data transfers, perfectly overlapping computation and communication, and introducing an upcoming transfer link standard called PCIE5.0.

Among the evaluated scenarios in our context, excluding those using the currently unavailable PCIE5.0, the application of pinned memory in addition to a perfect overlap of communication and computation, yields the best relative GPU decompression offloading performance, across all considered algorithms. For ZSTD, LZ4 and ZLIB this allows a GPU decompression offloading implementation to beat up till 8, 4 or 16 CPU threads, respectively[1]. Further performance improvements require the reduction of transfer time for ZSTD and LZ4, even after the introduction of PCIE5.0. On the other hand, further performance improvements for ZLIB require an improvement of the GPU decompression speed.

All in all, through the formulation, calibration and verification of latency models for the current execution of RDataFrame programs as well as two variants of its offloaded decompression, we could devise an efficient GPU decompression offloading approach, and predict its potential speedup for the LHCB benchmark, its datasets and our hardware across various compression algorithms. Performing a scenario analysis on the model predicting its maximum speedup, allowed us to identify better performing GPU decompression offloading implementation ideas for all considered algorithms, in the same context.

[1]For ZSTD and LZ4 this requires a high compression ratio.

Revisiting the main research question of this work, **"How can we optimise decompression in an application with the GPU?"**, this thesis thus demonstrates that *a model-based approach is a valid way to provide an answer.*

6.2 Contributions

This work provides a high-level view on the execution of RDataFrame programs in a single-threaded context, which was not readily available but scattered across sources such as documentation and code. Together with this high-level view, we created a novel latency model, for important components in the execution of RDataFrame programs. In the context of our LHCB benchmark, its dataset and our hardware, we found that a sub-model of this bigger model, can predict the single-threaded decompression latency accurately. From this sub-model, we also formulated a model that can provide accurate predictions for the multi-threaded decompression latency, in the same context. One can reuse the single-threaded decompression latency model for other applications with a columnar dataset, when the sum of the latency contribution per column type, modelled as its uncompressed size divided by a column-type dependent decompression speed, turns out as accurate. Additionally, one can reuse the multi-threaded model, when the application divides the decompression load evenly over all threads.

Given our knowledge about data decompression in RDataFrame programs, we provide and model multiple GPU decompression offloading ideas. Any such implementation consists of data transfers to and from the GPU, as well as decompression on the GPU. Our first idea was to transfer individual compressed pages to the GPU, for decompression, and then back to the CPU. Then we evaluated a second approach wherein we transfer buffers of compressed pages to the GPU for decompression, after which we transfer decompressed buffer counterparts back. All other approaches are incremental changes of this second approach, either using pinned memory in data transfers or also overlapping decompression and data transfers. One can reuse these GPU decompression offloading implementation ideas and models for other applications that use compressed data, as long as their datasets are also partitioned in segments alike pages in the RNTuple datasets.

Apart from offering reusable components that can make it easier to answer the research question for other applications with a model-based approach, this thesis also provides a reusable outline of such an approach. This outline consists of two steps. In the first step one should iteratively create a (partial) latency model for the application, from which it is possible to extract a model for the decompression latency contribution. As long as the

final model satisfies this criterion, earlier model iterations may contain terms that model more components than decompression alone. Additionally, one can also omit modelling application components, as long as they are unrelated to decompression. In the second step, one designs and models GPU decompression offloading approaches simultaneously. By comparing these models with each other, and with the model describing the current CPU decompression implementation, one can look for an approach that optimises the current implementation.

6.3 Limitations and Threats to Validity

For starters, one should note that the GPU decompression offloading speedup predictions presented in this work, only hold when using the LHCB benchmark on its own dataset type, and for our hardware. Fundamental to these speedup predictions, are the CPU decompression latency model and the GPU decompression latency models.

Because wanted to simplify modelling the CPU decompression latency as much as possible, we did not consider changing clock frequencies due to turbo boosting. In calibrating this model, we also fixed the mapping of RDataFrame threads, to separate cores. By default, this mapping is not set for RDataFrame programs. Actual mappings, as well as the usage of turbo boosting, will influence the CPU decompression latencies, in real execution scenarios.

Apart from modelling the CPU decompression latencies, we also provide various GPU decompression offloading latency models. In all approaches based on transferring buffers of pages, we do not consider the time required to fill these buffers. This makes the latency predictions of each corresponding model, conservative, and the predicted GPU decompression offloading speedups optimistic.

Remember that in calibrating the buffered GPU decompression offloading models, we determine relationships between the chunk size C and the decompression speed. For such a relationship and a specific C we actually measure the decompression speed by dividing a dataset in chunks of this size, for us to subsequently measure the average decompression speed across these chunks. This computation assumes that we can accurately characterise the decompression speed of a chunk of size C, using such a mean.

As final contributions of this work, we determined the influence of increasing compression ratio's in addition to scenario analyses. In both contributions, we assume that increasing compression ratio's do not influence the GPU decompression speeds. This might turn out as unrealistic because the compression ratio directly influences the amount of data that a

decompression algorithm needs to read, to produce a certain amount of uncompressed data. In the various scenario analyses, we also look at an implementation that perfectly overlaps computation and communication. Even though CUDA streams allow us to achieve such an overlap, we do not know whether a full overlap is feasible. This also begs the question insofar the corresponding speedup predictions are optimistic.

6.4 Future Work

On a high level, this work provides evidence that a model-based approach is a valid method to optimise decompression in an application using a GPU. Repeating this process for other applications, would provide more evidence for this claim.

Zooming in to the context of our case-study, future work could make speedup predictions for RDataFrame applications other than the LHCB benchmark and its dataset type, to evaluate the variation of speedups across this variable. Additionally, one could actually implement the suggested GPU decompression offloading approaches, to see how the speedup predictions hold up to reality.

Even though we showed that the presented GPU decompression offloading approaches, can beat a number of CPU threads, we did not see these approaches, convincingly outperform the decompression of data on a CPU. By extending the presented models to a multi-GPU environment, it could be possible to reach such a scenario. Another pathway is to consider the usage GPUs that share a memory space with the CPU, without requiring memory transfers, such as integrated GPUs.

At the end of the day, the goal of doing GPU decompression offloading is to minimise the time spent on decompression in any given node. Whereas GPU decompression offloading could not convincingly beat the CPU in this project, we might get lower GPU decompression offloading latencies by looking at other compression algorithms, specifically tailored for GPUs.

Finally note that in this work, we looked at optimising decompression in isolation. One could also widen the scope of optimisation with a GPU, to other or even all components of RDataFrame applications, as this might improve its performance as a whole.

7

Appendix

7.1 RunAndCheckFilters latency distributions

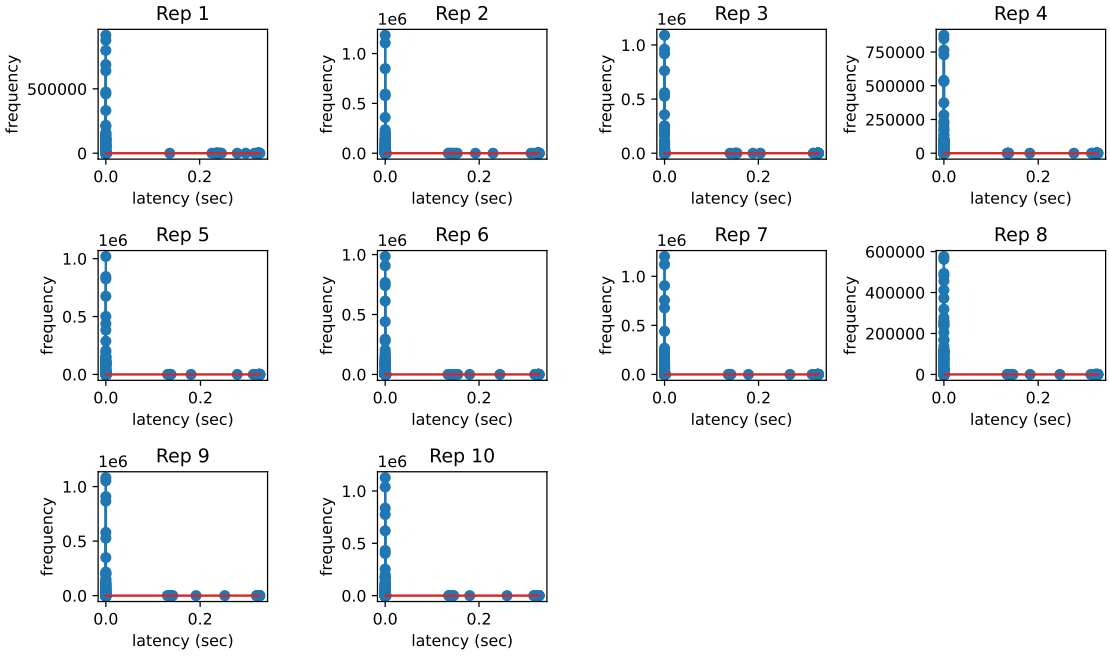


Figure 7.1: RunAndCheckFilter latency distributions for the LHCB benchmark, applied on the B2HHH dataset without compression, on a Amazon EC2 c4.8xlarge node.

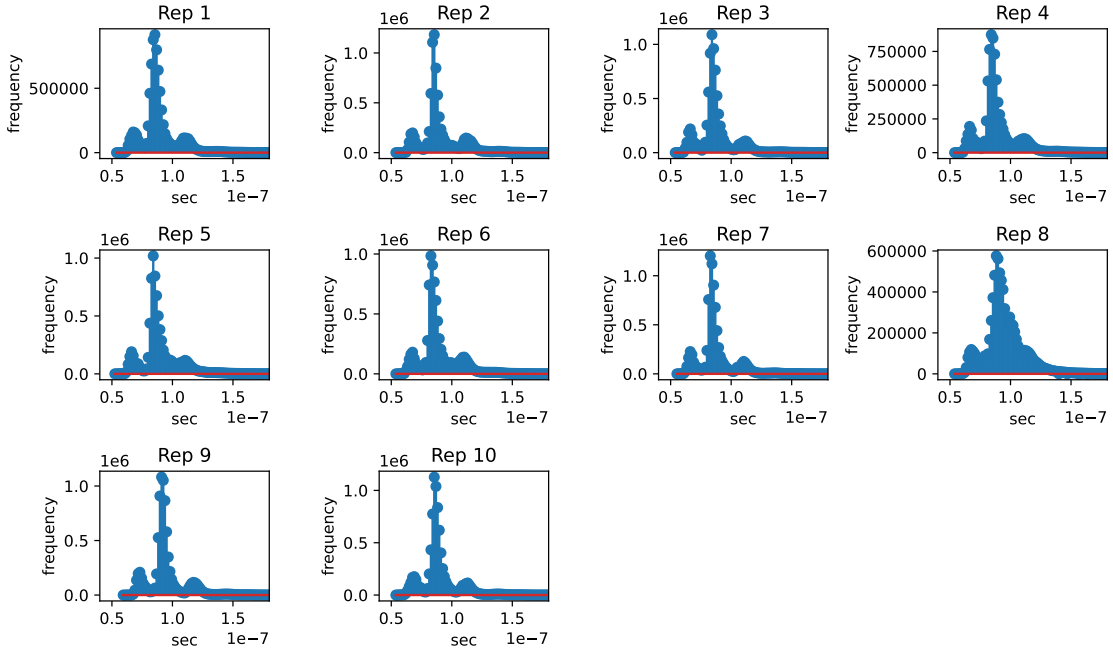


Figure 7.2: Contains the same contents as in Figure 7.1 for a different second range.

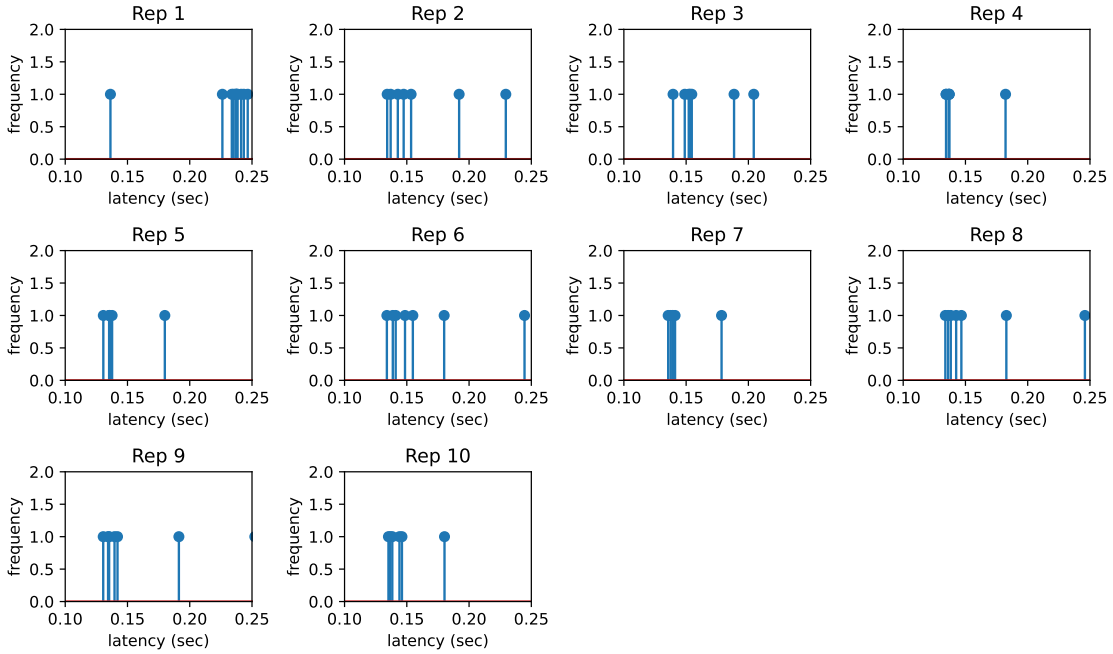


Figure 7.3: Contains the same contents as in Figure 7.1 for a different second range.

7.2 LHCB filter field fetching latency distribution segments

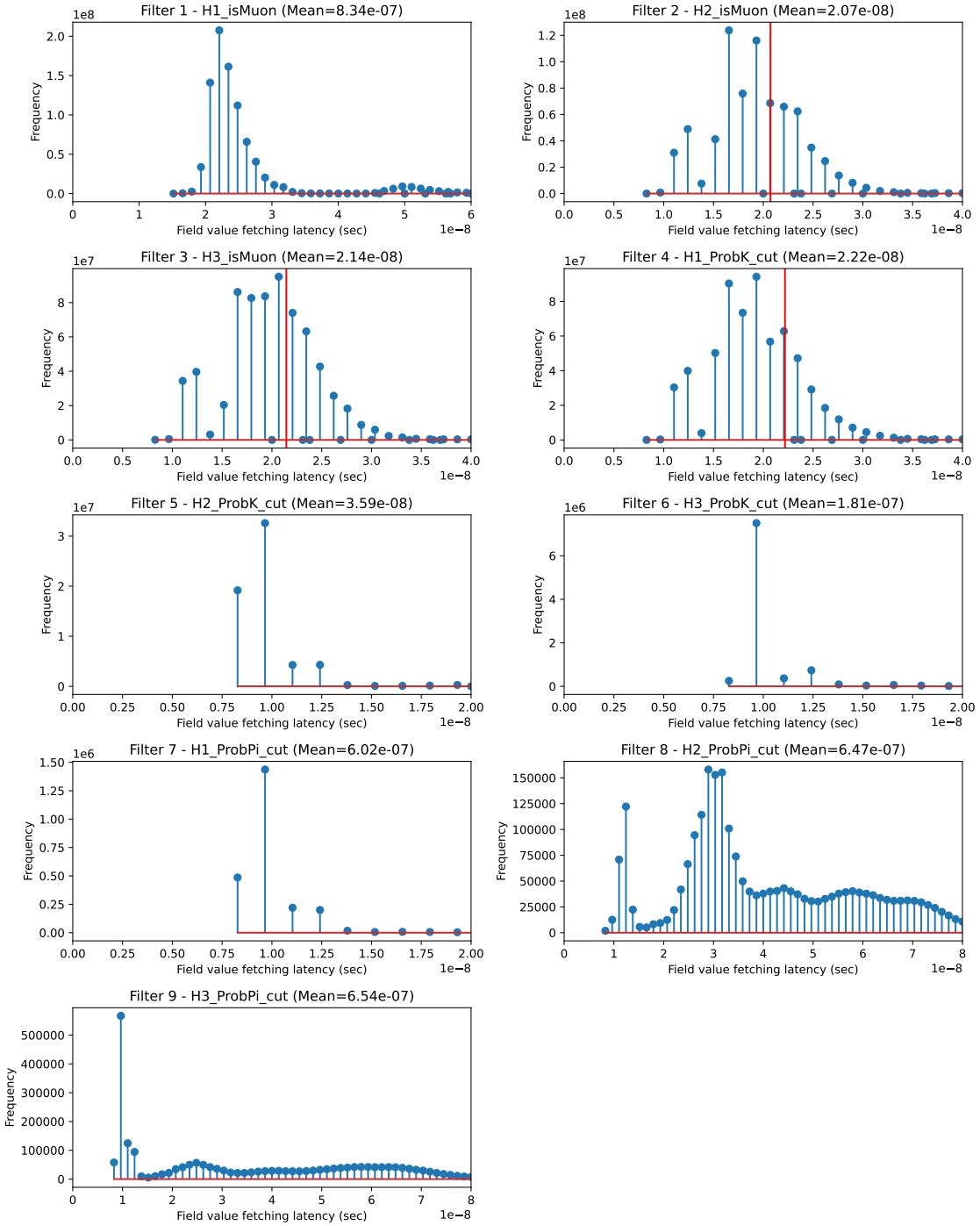


Figure 7.4: The low area of the field value fetching latency distribution for each filter in the LHCB benchmark, applied on the B2HHH dataset without compression (n=100, Amazon EC2 c4.8xlarge). Additionally, we display some distribution means using a red vertical line.

7.3 Convergence ratio simple linear models

Assume that we have two simple linear models $f(x) = ax + b$ and $g(x) = cx + d$. Then,

$$\begin{aligned}
\lim_{x\to\infty} \frac{f(x)}{g(x)} &= \lim_{x\to\infty} \frac{ax+b}{cx+d} \\
&= \lim_{x\to\infty} \left[\frac{ax}{cx+d} + \frac{b}{cx+d}\right] \\
&= \lim_{x\to\infty} \left[\frac{ax}{cx+d}\right] + \lim_{x\to\infty} \left[\frac{b}{cx+d}\right] \\
&= \lim_{x\to\infty} \left[\frac{ax}{cx+d}\right] + 0 \\
&= \lim_{x\to\infty} \left[\frac{a}{c+\frac{d}{x}}\right] \\
&= a/c
\end{aligned}$$

So the ratio of one such models with respect to another, converges to a constant value.

7.4 Transfer speed model - Strictly increasing

A function $f(x)$ is strictly increasing on its domain, whenever $f(x_2) > f(x_1)$ if $x_2 > x_1$. We want to proof that our transfer speed model of the form $f(x) = c - ae^{-b\sqrt{x}}$ is strictly increasing, if $a, b, c, x > 0$. In other words, $\forall x_2 > x_1 > 0 \wedge a, b, c > 0$, $c - ae^{-b\sqrt{x_2}} > c - ae^{-b\sqrt{x_1}}$. Assuming that this theorem is false, means that there exists $x_2 > x_1 > 0$ and $a, b, c > 0$ for which $c - ae^{-b\sqrt{x_2}} \leq c - ae^{-b\sqrt{x_1}}$. We can rewrite the latter inequality,

$$\begin{aligned}
c - ae^{-b\sqrt{x_2}} &\leq c - ae^{-b\sqrt{x_1}} \\
-ae^{-b\sqrt{x_2}} &\leq -ae^{-b\sqrt{x_1}} \\
ae^{-b\sqrt{x_2}} &\geq ae^{-b\sqrt{x_1}} \\
e^{-b\sqrt{x_2}} &\geq e^{-b\sqrt{x_1}} && (a > 0) \\
-b\sqrt{x_2} &\geq -b\sqrt{x_1} && (\log x \text{ strictly increases for } x > 0) \\
b\sqrt{x_2} &\leq b\sqrt{x_1} \\
\sqrt{x_2} &\leq \sqrt{x_1} && (b > 0) \\
x_2 &\leq x_1 && (x^2 \text{ strictly increases for } x > 0)
\end{aligned}$$

The final statement yields a contradiction with $x_2 > x_1 > 0$ which means the original theorem is true.

7.5 Node specifications

	DAS6 node	**CERN alma9 node**
CPU	AMD 7402P	AMD 7220P
GPU	NVIDIA A6000	-
Link GPU	PCIE 4.0x16	-
Main memory	8x 16GB @ 2933 MT/sec (HMA82GR7DJR8N-XN)	8x 16 GB @ 3200 MT/sec (M393A2K40DB3-CWE)
Secondary storage	?	MZWLJ3T8HBLS-00007

Table 7.1: Hardware specifications of nodes used for experiments in the thesis.

7.6 GPU offloading speedup predictions

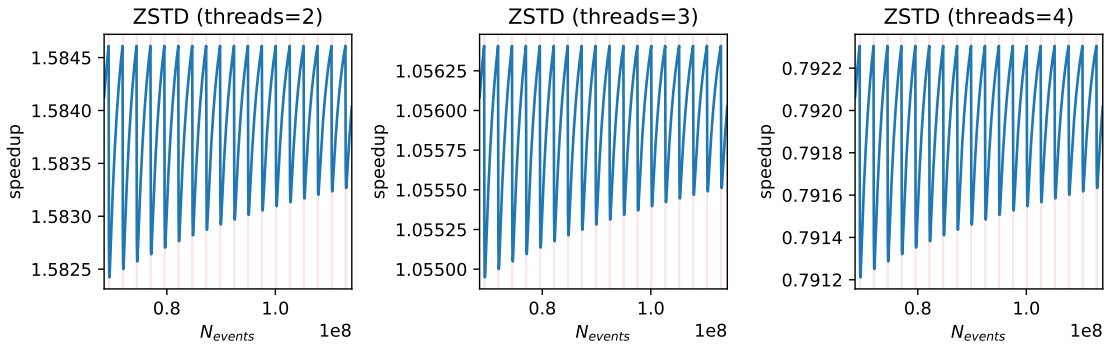


Figure 7.5: Displays the predicted GPU decompression offloading speedup with respect to the multi-threaded LHCB decompression time across various thread counts and sizes of a B2HHH dataset which uses ZSTD compression. Each graph displays the relation between dataset sizes and this predicted speedup, for a certain thread count. In each red area, inaccuracies in $\lambda^{\text{GPU}}_{\text{decompress}}$ cause inaccuracies in the speedup predictions.

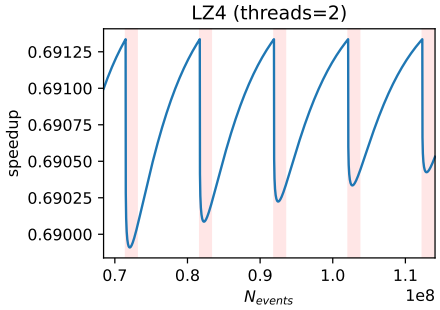


Figure 7.6: Displays the predicted GPU decompression offloading speedup with respect to the multi-threaded LHCB decompression time for 2 threads across various sizes of a B2HHH dataset which uses LZ4 compression. In each red area, inaccuracies in $\lambda^{\text{GPU}}_{\text{decompress}}$ cause inaccuracies in the speedup predictions.

References

[1] **Walmart Marketplace Statistics Sellers Should Know | Zentail Blog**. 1

[2] **How Walmart Puts AI and Machine Learning into Play | UC Davis Graduate School of Management**. 1

[3] Ian Baracskay, Donald J Baracskay III, Mehtab Iqbal, and Bart Piet Knijnenburg. **The diversity of music recommender systems**. In *27th International Conference on Intelligent User Interfaces*, pages 97–100, 2022. 1

[4] **Spotify — About Spotify**. 1

[5] **What is YouTube Music? Everything you need to know | Digital Trends**. 1

[6] **How Many Websites Are There in the World? (2023) - Siteefy**. 1

[7] **Number of Credit Card Transactions per Second & Year: 2023 Data**. 1

[8] **How Major Credit Card Networks Protect Customers Against Fraud | Bankrate**. 2

[9] **Card industry faces 400B in fraud losses over next decade, Nilson says | Payments Dive**. 2

[10] Peng Jiang, Sanju Sinha, Kenneth Aldape, Sridhar Hannenhalli, Cenk Sahinalp, and Eytan Ruppin. **Big data in basic and translational cancer research**. *Nature Reviews Cancer*, **22**(11):625–639, 2022. 2

[11] Joel E Pellot and Orlando De Jesus. **Diffuse Intrinsic Pontine Glioma**. In *StatPearls [Internet]*. StatPearls Publishing, 2023. 2

[12] Janika Raun, Rein Ahas, and Margus Tiru. **Measuring tourism destinations using mobile tracking data**. *Tourism Management*, **57**:202–212, 2016. 2

[13] Hiroaki Ogata, Misato Oi, Kousuke Mohri, Fumiya Okubo, Atsushi Shimada, Masanori Yamada, Jingyun Wang, and Sachio Hirokawa. **Learning analytics for e-book-based educational big data in higher education**. *Smart sensors at the IoT frontier*, pages 327–350, 2017. 2

[14] **High Energy Physics**. 2

[15] **Physics | CERN**. 2

[16] **Facts about the LHCB**. 2

[17] **The Large Hadron Collider | CERN**. 2

[18] **Experiments | CERN**. 2

[19] Ana Lopes and Melissa Loyse Perrey. **FAQ - LHC The guide**. Technical report, 2022. 2

[20] John Shalf. **The future of computing beyond Moores Law**. *Philosophical Transactions of the Royal Society A*, **378**, 3 2020. 2

[21] Oksana Shadura and B Bockelman. **ROOT I/O compression algorithms and their performance impact within Run 3**. In *Journal of Physics: Conference Series*, **1525**, page 012049. IOP Publishing, 2020. 3

[22] Max Plauth and Andreas Polze. **GPU-Based decompression for the 842 algorithm**. *Proceedings - 2019 7th International Symposium on Computing and Networking Workshops, CANDARW 2019*, pages 97–102, 11 2019. 3

[23] **Adaptive loss-less data compression method optimized for GPU decompression**. *Concurrency and Computation: Practice and Experience*, **29**:e4283, 12 2017. 3

[24] **Massively parallel ANS decoding on GPUs**. *ACM International Conference Proceeding Series*, 8 2019. 3

[25] Shaurya Patel, Tongping Liu, and Hui Guan. **FreeLunch: Compression-based GPU Memory Management for Convolutional Neural Networks**. In *2021 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 1–8. IEEE, 2021. 3

[26] Jingcheng Shen, Yifan Wu, Masao Okita, and Fumihiko Ino. **Accelerating GPU-Based Out-of-Core Stencil Computation with On-the-Fly Compression**. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **13148 LNCS**:3–14, 2022. 3

[27] Mo Sha, Yuchen Li, and Kian Lee Tan. **GPU-based graph traversal on compressed graphs**. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 775–792, 6 2019. 3

[28] Jing Li, Hung Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. **HippogriffDB**. *Proceedings of the VLDB Endowment*, **9**:1647–1658, 10 2016. 3

[29] Yang Liu, Jianguo Wang, and Steven Swanson. **Griffin: Uniting CPU and GPU in information retrieval systems for intra-query parallelism**. *ACM SIGPLAN Notices*, **53**(1):327–337, 2018. 3

[30] **ROOT Primer**. 4

[31] Jürgen Gross. *Linear regression*, **175**. Springer Science & Business Media, 2003. 6

[32] **LHCB benchmark code**. 9

[33] **ROOT: TTree Class Reference**. 9

[34] **ROOT: RNTuple Introduction**. 9

[35] **ROOT::RDataFrame Class Reference**. `Rdataframehttps://root.cern/doc/master/classROOT_1_1RDataFrame.html`. 9, 38

[36] **B2HHH dataset**. 12

[37] **gen_lhcb : converts B2HHH format from TTree to RNTuple**. 12

[38] **root/tree/dataframe/src/RLoopManager.cxx**. 14

[39] **root/tree/dataframe/src/RNTupleDS.cxx - GetEntryRanges**. 15

[40] **root/tree/dataframe/src/RNTupleDS.cxx - Finalize**. 15

[41] **root/tree/dataframe/inc/ROOT/RNTupleDS.hxx - Parent class declaration**. 15

[42] **root/tree/dataframe/inc/ROOT/RDataSource.hxx - InitSlot**. 15

[43] **root/tree/dataframe/inc/ROOT/RDataSource.hxx - FinalizeSlot**. 15

[44] **root/tree/dataframe/src/RNTupleDS.cxx**. 15

[45] **root/core/base/inc/TStopwatch.h**. 15

[46] **std::chrono::system_clock - cppreference.com**. 15

[47] **std::unordered_map - cppreference.com**. 15

[48] **root/tree/dataframe/inc/ROOT/RDF/RInterface.hxx - Report**. 17

[49] **root/root/tree/dataframe/src/RNTupleDS.cxx - SetEntry**. 18

[50] **root/tree/dataframe/inc/ROOT/RDF/RLoopManager.hxx - fBookedActions**. 19

[51] **ROOT: ROOT::Internal::RDF::RActionBase Class Reference - ROOT 6.30**. 19

[52] **ROOT: ROOT::Internal::RDF::RActionBase Class Reference - ROOT 6.28**. 19

[53] **root/tree/dataframe/inc/ROOT/RDF/RInterface.hxx - Vary**. 19

[54] **ROOT: ROOT::RDataFrame Class Reference - Jitting**. 19

[55] **ROOT: ROOT::Detail::RDF::RFilterBase Class Reference - ROOT 6.30**. 19

[56] **ROOT: ROOT::Detail::RDF::RFilterBase Class Reference - ROOT 6.28**. 19

[57] **root/tree/dataframe/inc/ROOT/RDF/RLoopManager.hxx - fBookedNamedFilters**. 19

[58] **ROOT: ROOT::Experimental::RNTupleDescriptor Class Reference**. `https://root.cern/doc/master/classROOT_1_1Experimental_1_1RNTupleDescriptor.html`. 27

[59] **ROOT: tree/ntuple/v7/src/RClusterPool.cxx Source File - ExecReadClusters**. https://root.cern.ch/doc/master/RClusterPool_8cxx_source.html#l00107. 28

[60] **ROOT::Experimental::Detail::RPageSource Class Reference - LoadClusters**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1RPageSourceFile.html#a049307ba5600a4c34228b9a649fa98be. 28

[61] **ROOT::Experimental::Detail::RClusterPool::RReadItem Struct Reference**. https://root.cern.ch/doc/master/structROOT_1_1Experimental_1_1Detail_1_1RClusterPool_1_1RReadItem.html. 28

[62] **ROOT: ROOT::Experimental::Detail::RPageSource Class Reference**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1RPageSource.html. 28

[63] **ROOT::Experimental::Detail::ROnDiskPage Class Reference**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1ROnDiskPage.html. 28

[64] **ROOT: ROOT::Experimental::Detail::ROnDiskPageMap Class Reference**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1ROnDiskPageMap.html. 28

[65] **ROOT: ROOT::Experimental::Detail::RClusterPool::RUnzipItem Struct Reference**. https://root.cern/doc/master/structROOT_1_1Experimental_1_1Detail_1_1RClusterPool_1_1RUnzipItem.html. 28

[66] **ROOT: ROOT::Experimental::Detail::RClusterPool Class Reference - ExecUnzipClusters**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1RClusterPool.html#a1bf262288e443225f290d9b9f292634e. 29

[67] **ROOT: ROOT::Experimental::Detail::RPageSource Class Reference - UnzipCluster**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1RPageSource.html#ad185c9373a030adfcf4a7de0cbf73d02. 29

[68] **ROOT: ROOT::Experimental::Detail::RPageAllocatorFile Class Reference**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1RPageAllocatorFile.html. 29

[69] **ROOT: ROOT::Experimental::Detail::RPageAllocatorDaos Class Reference**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1RPageAllocatorDaos.html. 29

[70] **ROOT: ROOT::Experimental::Detail::RPageSourceFriends Class Reference**. https://root.cern/doc/master/classROOT_1_1Experimental_1_1Detail_1_1RPageSourceFriends.html. 29

[71] DAOS Project. **Welcome to DAOS Version 2.0! - DAOS v2.0**. https://docs.daos.io/v2.0/. 29

[72] **ROOT: ROOT::Experimental::Detail::RPageSource Class Reference - UnzipClusterImpl**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1RPageSource.html#abfc4b2d2c6e2632a81b45497fcde492e. 29

[73] **ROOT: tree/ntuple/v7/src/RPageStorage.cxx Source File - UnsealPage**. https://root.cern.ch/doc/master/RPageStorage_8cxx_source.html#l00148. 29

[74] **ROOT: ROOT::Experimental::Detail::RPage Class Reference**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1RPage.html. 29

[75] **ROOT: ROOT::Experimental::Detail::RPagePool Class Reference**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1Detail_1_1RPagePool.html. 29

[76] **ROOT: tree/ntuple/v7/src/RClusterPool.cxx Source File - GetCluster**. https://root.cern.ch/doc/master/RClusterPool_8cxx_source.html#l00242. 30

[77] **NTupleInfo program**. https://github.com/jblomer/iotools/blob/master/ntuple_info.C. 37

[78] **ROOT: tree/ntuple/v7/src/RNTupleDescriptorFmt.cxx Source File**. https://root.cern.ch/doc/master/RNTupleDescriptorFmt_8cxx_source.html. 37

[79] **ROOT: ROOT::Experimental::RNTupleDS Class Reference - GetEntryRanges**. https://root.cern.ch/doc/master/classROOT_1_1Experimental_1_1RNTupleDS.html#a8295a959e7e27a747951fdd3845515fb. 38

[80] **ROOT: tree/dataframe/src/RLoopManager.cxx Source File - RunDataSourceMT**. https://root.cern.ch/doc/master/RLoopManager_8cxx_source.html#l00613. 38

[81] **root/tree/ntuple/v7/inc/ROOT/RPageStorageFile.hxx - RPageSoureFile**. 48

[82] **root/tree/ntuple/v7/src/RPageStorageFile.cxx - PopulatePageFromCluster**. 48, 65, 69

[83] **PCIe Gen 4 vs. Gen 3 Slots, Speeds**. 56, 93

[84] **NVCOMP**. 64

[85] **GitHub - NVIDIA/nvcomp**. 64

[86] **ZLIB Home site**. 64

[87] **nvcomp/doc/highlevel_cpp_quickstart.md**. 64

[88] **Accelerating Lossless GPU Compression with New Flexible Interfaces in NVIDIA nvCOMP | NVIDIA Technical Blog**. 64

[89] **nvcomp/doc/lowlevel_c_quickstart.md**. 65

[90] **root-gpu-decompression/gpu_root_decomp.cu**. 66

[91] **nvcomp/examples/low_level_quickstart_example.cpp**. 69

[92] W Hwu Wen-Mei, David B Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022. 90

[93] **How to Overlap Data Transfers in CUDA C/C++ | NVIDIA Technical Blog**. 93

[94] **7.1 Annotated Slides | Computation Structures | Electrical Engineering and Computer Science | MIT OpenCourseWare**. 93