GPU-based Online Processing for the ALICE and CBM Experiments

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich 12

der Johann Wolfgang Goethe-Universität

in Frankfurt am Main

von

Felix Weiglhofer

aus Frankfurt

Frankfurt (2025)

(D 30)

Vom Fachbereich 12 der

Johann Wolfgang Goethe - Universität als Dissertation angenommen.

Dekan: Prof. Dr. Bastian von Harrach-Sammet

Gutachter: Prof. Dr. Volker Lindenstruth, Prof. Dr. Ivan Kisel

Datum der Disputation:

Abstract

Modern high-energy physics (HEP) experiments produce data at unprecedented scales, necessitating real-time processing solutions capable of handling throughput exceeding 1 TB/s. This dissertation addresses these challenges by developing and deploying GPU-accelerated workflows in the ALICE and CBM experiments at CERN and FAIR, respectively.

In ALICE, several components of the Time Projection Chamber (TPC) reconstruction chain were optimized for GPUs, including the initial zero-suppression decoding, a parallelized track merger yielding a 30x speedup, and an efficient gathering kernel for the final data transfers. These optimizations enable the O^2 framework to sustain real-time event reconstruction at collision rates up to 50 kHz, handling the full 1 TB/s data stream since the start of LHC Run 3 in 2022.

Furthermore xpu is introduced, a lightweight C++ library designed for portable GPU programming across CUDA, HIP, and SYCL backends. By exploiting separate compilation of device code and dynamic backend selection, xpu achieves near-native performance on multiple architectures with negligible overhead. The library is designed for modern C++ without sacrificing performance or predictability.

Finally, a complete GPU-based reconstruction chain for the Silicon Tracking System (STS) written in xpu for CBM was developed and integrated into the experiment's free-streaming data framework. The GPU implementation achieves up to 122x speedup over the previous CPU-based code by employing parallel algorithms for cluster finding, hit reconstruction, and an optimized merge sort. When deployed during the May 2024 mCBM beamtime, the system reliably processed up to 2.4 GB/s, demonstrating the feasibility of high-rate online data processing for future CBM operations.

Contents

Chapter 1

Introduction

Modern high-energy physics (HEP) experiments face unprecedented computational challenges due to increasing collision rates and detector complexity. At the Large Hadron Collider (LHC), traditional triggered readout systems are being replaced by continuous data acquisition to enable more sophisticated event selection. In this context, two different processing phases are used: *online processing*, which refers to real-time data analysis during detector operation under strict timing constraints, and *offline processing*, which encompasses the analysis of stored data after data taking with less stringent time requirements but often more sophisticated algorithms. The ALICE experiment pioneered the online processing approach for Run 3, processing heavy-ion collisions at 50 kHz without a hardware trigger [6], producing data rates exceeding 1 TB/s. Similarly, LHCb has moved to a software-only trigger system [59], while the upcoming CBM experiment at FAIR plans continuous readout at collision rates up to 10 MHz for heavy-ion collisions with expected data rates around 500 GB/s [2].

Graphics Processing Units (GPUs) have proven to be particularly well-suited for HEP reconstruction algorithms like track finding and cluster reconstruction, making it feasible to handle these extreme data rates in soft real-time. The ALICE High Level Trigger first demonstrated large-scale GPU usage in high-energy physics, employing GPUs for real-time track reconstruction during LHC Run 2 [9]. For Run 3, LHCb developed the Allen GPU trigger system that processes the full 40 Tbit/s detector readout [4]. ALICE's new O^2 framework uses GPUs for the majority of the online processing and to speed up offline processing, with the complete TPC reconstruction running on GPUs to handle up to 50 kHz Pb–Pb data [33]. While these implementations demonstrate the performance potential of GPUs in HEP, the decade-plus operational lifetimes of these experiments create significant challenges for GPU software development. The code must not only support multiple hardware architectures for initial deployment but also remain maintainable as GPU technology evolves and hardware is upgraded over the experiment's lifetime.

This dissertation presents three major contributions to GPU computing in high-energy physics experiments. For the ALICE experiment, several components throughout the TPC reconstruction chain were optimized for GPU processing, including zero-suppression decoding of raw detector data, parallelized track merging providing 30x speedup over the sequential implementation and an efficient gathering kernel that optimizes

the final DMA transfer of clusters to the host. The second contribution introduces xpu, a lightweight C++ library for portable GPU programming that enables writing hardware-independent code while maintaining native performance with neglible overhead. Through separate compilation of device code and dynamic backend selection, xpu supports multiple GPU architectures while allowing optimal compiler usage for each platform. For the CBM experiment, a complete GPU-accelerated reconstruction chain for the Silicon Tracking System (STS) was developed. Compared to the existing CPU-based reconstruction, the implementation achieves a 122x speedup through parallel algorithms for cluster finding and hit reconstruction and a custom merge sort implementation optimized for GPUs. The STS reconstruction is tested alongside the other CBM online components under real-world conditions in synthetic runs as well as during a four-day beamtime in the mini-CBM test setup.

The following chapters discuss these contributions in more detail:

- Chapter 2 provides background on the ALICE and CBM experiments, with focus on the Time Projection Chamber and Silicon Tracking System detectors. The chapter also describes the computing infrastructure used for data processing in both experiments.
- Chapter 3 examines various GPU processing tasks in the ALICE TPC reconstruction chain. This includes optimizations for zero suppression decoding, cluster gathering, and track merging, along with a detailed performance analysis of these improvements.
- Chapter 4 presents xpu, a lightweight C++ library for portable GPU programming. The chapter discusses the design principles, implementation details, and performance characteristics of the library, demonstrating how it enables efficient cross-platform GPU code.
- Chapter 5 describes the development of a GPU-accelerated reconstruction chain for CBM's Silicon Tracking System. The implementation of parallel algorithms for cluster finding and hit reconstruction is detailed, followed by a performance evaluation using both simulated and real detector data.
- Chapter 6 summarizes the contributions and briefly discusses future directions for GPU computing in high-energy physics experiments.

Chapter 2

Background

This chapter provides background information about the two high-energy physics experiments that provide the context for the development of the software and algorithms discussed in this thesis: the ALICE experiment at CERN and the Compressed Baryonic Matter (CBM) experiment at FAIR.

Section 2.1 introduces the ALICE experiment at CERN's Large Hadron Collider, with particular focus on its Time Projection Chamber (TPC) detector (Section 2.1.1) and the computing infrastructure used for data processing in Run 3. Section 2.1.2 describes the Event Processing Node (EPN) farm that handles the substantial data rates from Pb–Pb collisions and Section 2.1.3 outlines the O^2 software framework used for data processing in ALICE.

Section 2.2 presents the upcoming CBM experiment at FAIR, focusing on its Silicon Tracking System (STS) detector (Section 2.2.1) and the challenges posed by its high interaction rates and triggerless data acquisition. Section 2.2.2 covers the mCBM test setup at GSI, which serves as a testbed for both detector hardware and online software development.

2.1 The ALICE Experiment

The ALICE (A Large Ion Collider Experiment) experiment [5] is one of the four major experiments at the Large Hadron Collider (LHC) at CERN. Unlike the other LHC experiments, that primarily study proton-proton collisions, ALICE is specifically designed to study heavy-ion collisions. Its main goal is to investigate the properties of the quark-gluon plasma, a state of matter that existed microseconds after the Big Bang where quarks and gluons move freely before combining into hadrons.

ALICE is a complex detector system consisting of multiple specialized components, as shown in Figure 2.1. At its core lies the Time Projection Chamber (TPC), a large cylindrical detector used for tracking charged particles and particle identification. The TPC is surrounded by several other detectors including the Inner Tracking System (ITS) for

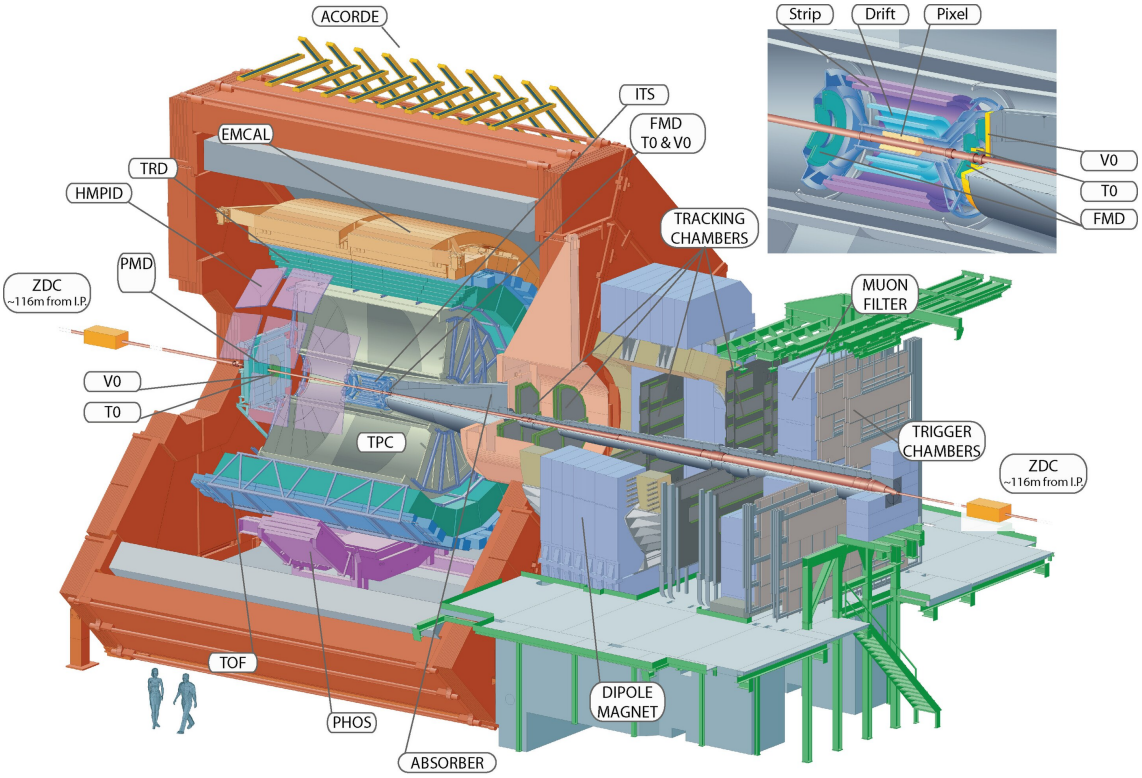

Figure 2.1: Schematic representation of the ALICE experiment. (Image source: [10])

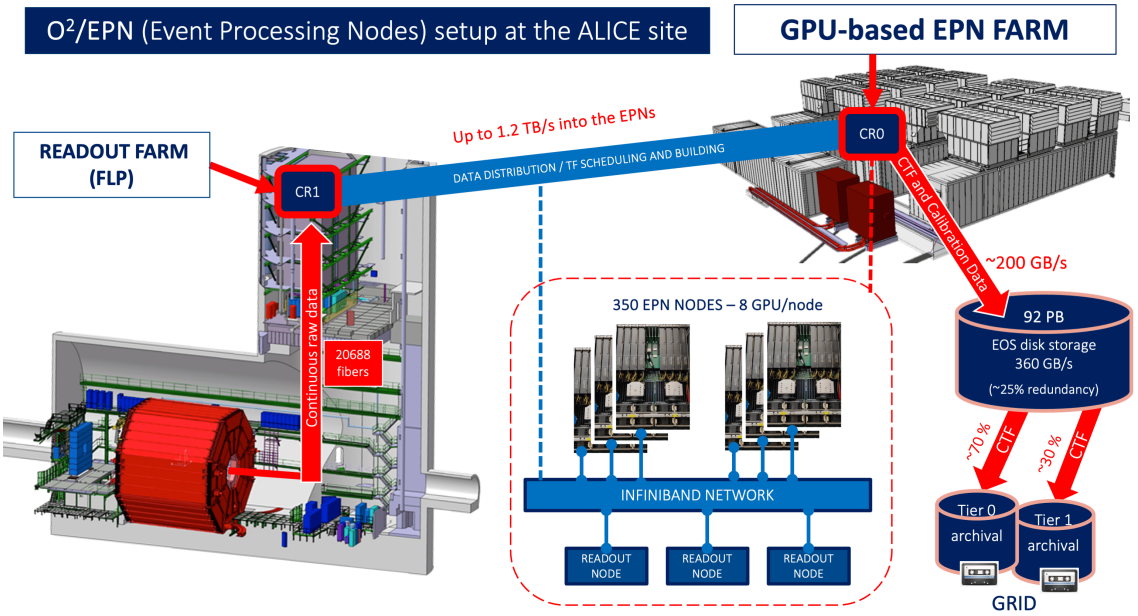

Figure 2.2: Illustration of the dataflow in ALICE. (Image source: [57])

precise vertex determination, the Time-Of-Flight detector (TOF) for particle identification, and various forward detectors for event characterization and triggering.

The experiment has undergone periodic upgrades to improve its capabilities. During Long Shutdown 2 (2019–2022), ALICE received major upgrades to handle increased collision rates during LHC Run 3, including a new continuous readout system and enhanced computing infrastructure [6]. This enables the experiment to record Pb–Pb collisions at rates up to 50 kHz and process the resulting data stream of up to 1.2 TB/s in real time.

2.1.1 The TPC Detector

The Time Projection Chamber (TPC) [12] is the main tracking detector of ALICE. The detector is designed as a large cylindrical chamber, 5 meters in length and 5 meters in diameter, filled with a gas mixture. When charged particles pass through the TPC, they ionize the gas along their path, creating trails of electrons that drift towards the endcaps under the influence of an electric field. At the endcaps, these signals are amplified and detected, allowing the reconstruction of particle trajectories in three dimensions with high precision.

In addition to tracking, the TPC also plays a crucial role in particle identification by measuring the energy loss of particles as they pass through the gas. The detector underwent a major upgrade during the Long Shutdown 2 to cope with the increased

collision rates in Run 3. The original readout chambers were replaced with more modern detectors based on Gas Electron Multiplier (GEM) technology, enabling the TPC to handle interaction rates of up to 50 kHz [25].

2.1.2 The ALICE EPN Farm

The Event Processing Node (EPN) farm forms a crucial part of ALICE's upgraded computing infrastructure for Run 3. The farm consists of 350 servers equipped with 2800 GPUs in total, designed to handle the substantially increased data rates from Pb–Pb collisions at up to 50 kHz interaction rates.

The computing nodes are divided into two configurations: older nodes equipped with AMD MI50 GPUs and newer nodes featuring AMD MI100 GPUs. The MI50 nodes contain two AMD EPYC 7452 32-core CPUs and 512 GB of DDR4-3200 RAM per node, while the MI100 nodes are equipped with two AMD EPYC 7552 48-core CPUs and 1 TB of DDR4-3200 RAM. Each GPU has 32 GB of main memory.

Processing of Time Projection Chamber (TPC) data presents the main computational challenge, requiring around 90 % of the GPU compute capacity during data taking. The use of GPUs was driven by both efficiency and cost considerations — a CPU-only solution would require approximately eight times as many servers to achieve equivalent processing power.

The EPN farm is housed in modular IT containers at the LHC Point 2 experiment site. Three of these containers currently host the worker nodes, infrastructure nodes, and network equipment. The facility employs an energy-efficient adiabatic cooling system that helps reduce operational costs and environmental impact. Under typical operating conditions during Pb–Pb data taking, the farm consumes around 550 kW of power while maintaining a Power Usage Effectiveness (PUE) ratio as low as 1.05.

Data flows from the detector readout electronics through First Level Processor (FLP) nodes near the experimental cavern to the EPN farm via a high-speed InfiniBand network, as illustrated in Figure 2.2. The FLP nodes can transfer data at rates up to 1.2 TB/s to the EPN farm for processing. After reconstruction and compression, the data is transferred at around 200 GB/s to CERN's central IT storage facilities (EOS) that provide approximately 92 PB of storage with 25 % redundancy. From there, the data is distributed to archival storage on the computing grid, with about 70 % going to Tier 0 and 30 % to Tier 1 storage. The modular design of the farm allows for future upgrades, with plans already in place to expand capacity during Long Shutdown 3 to handle the anticipated 20 % increase in data rates for Run 4. A full description of the EPN farm is given in [58].

2.1.3 The O^2 Framework

The O^2 framework is the software package of ALICE for data processing in Run 3. It unifies what were previously separate online and offline frameworks into a single system that handles both synchronous (real-time) and asynchronous data processing. The framework is built on a message-passing architecture using the FairMQ library [71] as its transport layer. Processing is organized into separate processes called devices that can be connected as different topologies to form processing workflows.

The framework employs a layered architecture: The transport layer handles data movement between processes, with optimizations like shared memory for local communication. A data layer provides different serialization backends including ROOT [15] as well as detector-specific formats optimized for GPU processing. The top-level Data Processing Layer (DPL) presents users with a high-level interface where they can describe their processing as a directed graph of tasks with data dependencies.

A key feature of O^2 is its ability to handle both synchronous processing during data taking, where data must be processed in real-time to keep up with detector readout, and asynchronous processing for final reconstruction. The same code and workflows can be used in both scenarios, with only the data source changing between live detector data and stored compressed data. The framework provides automatic deployment capabilities for distributed processing but can also run workflows locally on a single machine for development and debugging. A full description of the O^2 framework is given in [33].

2.2 The Compressed Baryonic Matter Experiment

The Compressed Baryonic Matter (CBM) experiment [60] is being built as part of the Facility for Antiproton and Ion Research (FAIR) [39] at the GSI Helmholtz Centre for Heavy Ion Research [41]. The goal of the experiment is to explore the properties of dense nuclear matter, similar to what exists in neutron stars and in the core of supernova explosions. To this end, heavy-ion collisions are created to study this dense matter. Data-taking is currently expected to begin in 2028.

CBM is expected to run at very high collision rates of up to 10 MHz. At these rates, it is no longer feasible to select potentially interesting events with a hardware trigger. Instead a so called ”triggerless” or ”free-streaming” data acquisition is planned with event selection being done in software. This allows for detecting rare probes and more complex decay topologies that might be missed by a hardware trigger. However, this requires an efficient online processing that reconstructs the raw data back into particle tracks. Online processing will take place in the Green IT Cube data center while data acquisition will be performed in the FLES entry cluster located near the CBM cave. Both clusters will be connected via a long-range InfiniBand network. Figure 2.3 shows a rendering of the planned extension of the GSI campus with the CBM cave and Green

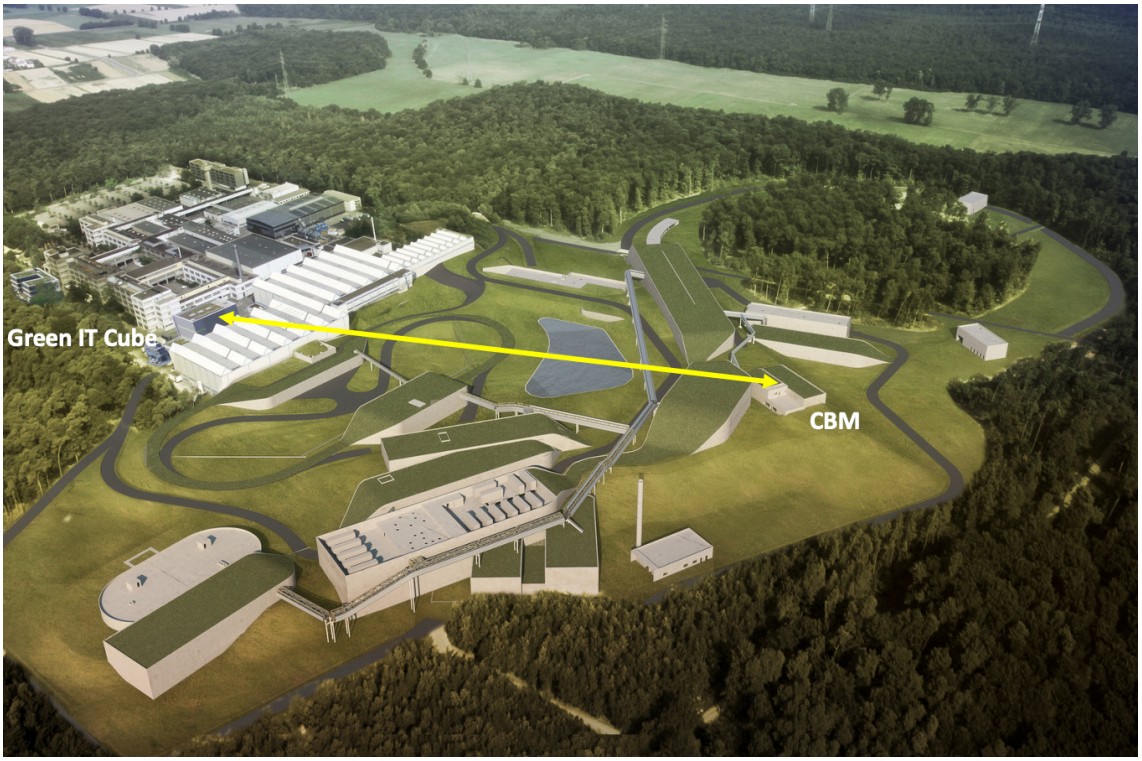

Figure 2.3: Rendering of the GSI compound with the finished SIS-100 extension. Marked are the CBM building and the Green IT cube where the online processing will take place. (Image source: [2])

IT cube. The planned setup is described in detail in the CBM online Technical Design Report [2].

2.2.1 The Silicon Tracking System Detector

The Silicon Tracking System (STS) detector is a central component of the CBM experiment. Positioned inside the magnet and upstream of the target, it is the first detector after the Beam Monitor together with the MVD. Made up of double-sided silicon strip sensors spread across 8 layers (also referred to as stations) it enables track reconstruction and momentum determination of charged particles. The planned detector layout is shown in figure 2.5. Stations are made up of double-sided strip sensors[1] consisting of 8 ASICs with 128 readout channels each on the front and back sides. Strips on the front side run vertically, while backside strips are rotated by 7.5°. Figure 2.6 shows a single sensor with readout cables. In total the STS will be made up of 896 of these sensors. The high temporal and spatial resolutions of 5 ns and 25 µm make it a key de-

[1] Also referred to as *modules* within the CBM software.

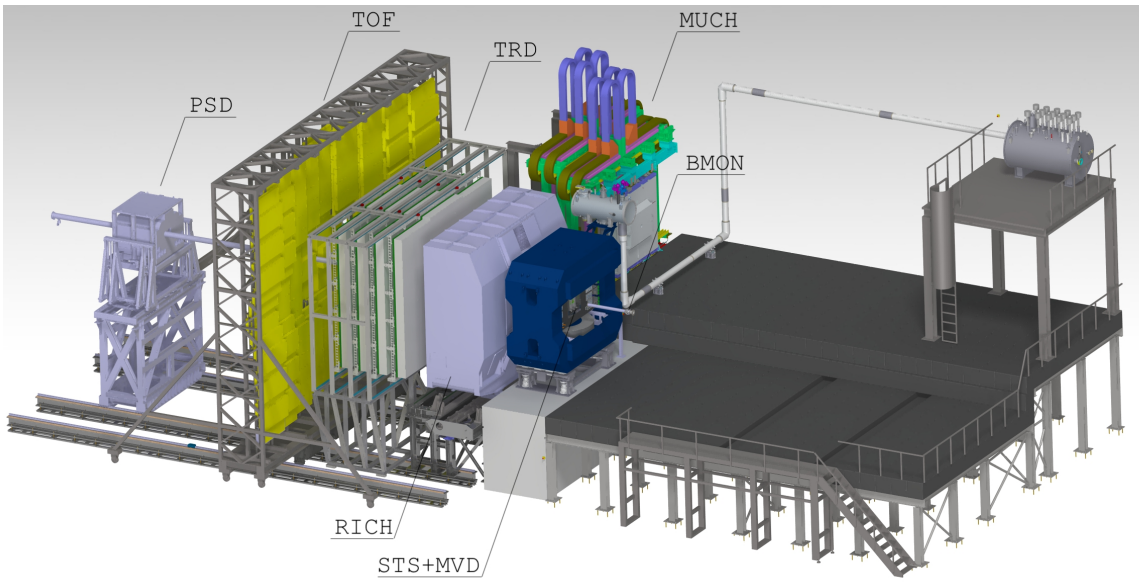

Figure 2.4: Schematic representation of the CBM experiment.
(Image source: [2])

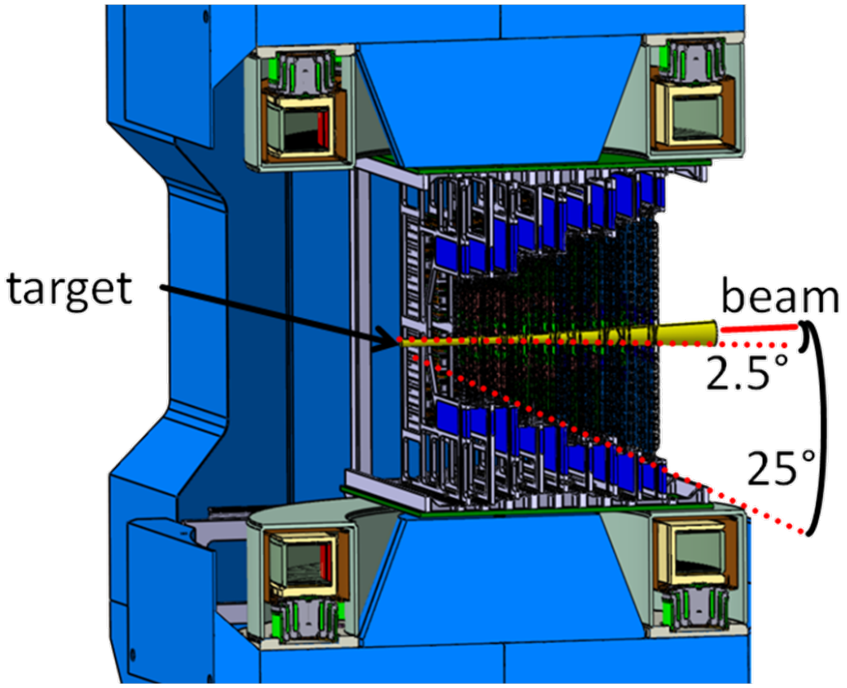

Figure 2.5: View of the STS stations placed inside the dipol magnet.
(Image source: [3])

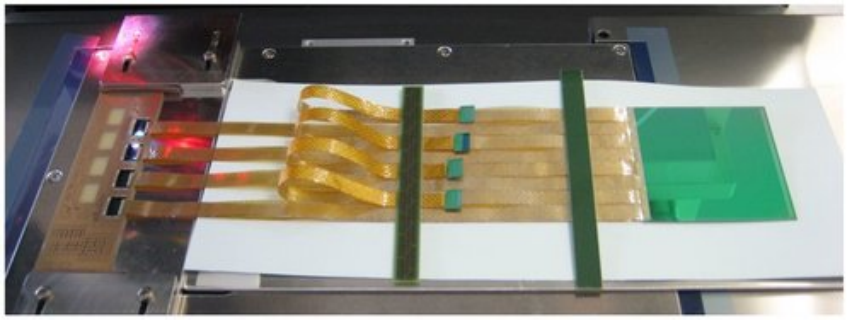

Figure 2.6: A sensor of the STS detector connected to readout cables. (Image source: [61])

tector for track reconstruction and thus particularly important for online processing. A full description of the system is given in the associated Technical Design Report [3].

2.2.2 The mCBM Experiment

The mCBM experiment is a FAIR Phase-0 experiment that is also used as a proof of concept for the CBM hardware [17]. It has been connected to the beam of the SIS18 synchrotron [64] and has been in operation since 2018. The system consists of small prototypes for all major subsystems. As such a mSTS detector is installed with 2 tracking stations comprising of 11 modules. A third station, dubbed *Station 0*, with a single module was installed in 2023 in front of the other two at 5 cm upstream of the target. Figure 2.7 shows a photo of the mCBM setup.

The mCBM experiment is connected to an mFLES system that supports the free-streaming data acquisition. This creates the opportunity to use it as a test environment for the CBM online software alongside the detector hardware. The online reconstruction of mCBM data is discussed in detail in Chapter 5.

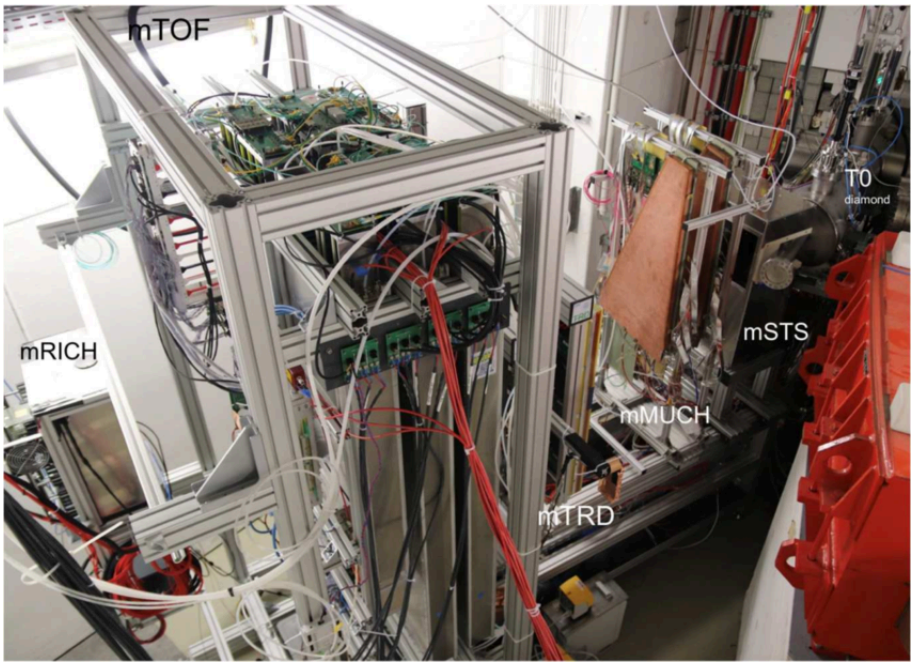

Figure 2.7: The mCBM experiment setup at SIS18. (Image source: [34])

Chapter 3

Online GPU Processing in ALICE

3.1 Introduction

This chapter examines various GPU processing tasks in ALICE's Time Projection Chamber (TPC) reconstruction chain, focusing on improvements and optimizations implemented for Run 3. The TPC, introduced in Section 2.1.1, serves as ALICE's main tracking detector and generates the majority of data that must be processed.

The collision rates increased in Run 3, reaching up to 50 kHz in Pb–Pb collisions compared to at most 8 kHz during Run 2 [6], require real-time processing of data rates exceeding 1 TB/s. This necessitated the development of a GPU-accelerated reconstruction chain running on the Event Processing Node (EPN) farm detailed in Section 2.1.2. Central to this processing chain is the TPC cluster finder, which combines charge deposits in neighboring pads and time bins into clusters representing particle crossing points. Section 3.2 provides an overview of this algorithm's key steps. This is meant to provide context for the rest of the chapter, as most of the discussed components are adjacent or related to the cluster finder.

The remainder of this chapter is structured as follows:

- Section 3.3 describes the fragment-based processing strategy that enables the cluster finder to handle arbitrary-length time frames within GPU memory constraints.
- Section 3.4 examines the implementation of zero-suppressed data decoding on GPUs, comparing the performance characteristics of different encoding formats.
- Section 3.5 details the implementation of the noisy pad filter, which identifies and excludes malfunctioning TPC pads that could interfere with cluster finding.
- Section 3.6 discusses the propagation of Monte Carlo labels through the reconstruction chain, essential for evaluating detector and reconstruction performance using simulated data.
- Section 3.7 presents improvements to the track merger, focusing on parallel processing techniques that significantly reduce execution time.

- Section 3.8 explores optimizations in cluster gathering, where efficient data movement between host and device memory is crucial for overall system performance.

All components discussed in this chapter are part of ALICE's online processing software in the O^2 framework and have been successfully used for data taking since the start of Run 3 in 2022.

3.1.1 Test Setup

All performance measurements presented in this chapter were conducted on the ALICE Event Processing Node (EPN) farm. The tests used both generations of processing nodes currently deployed in the farm: the older nodes equipped with AMD MI50 GPUs and newer nodes featuring AMD MI100 GPUs. The MI50 nodes contain two AMD EPYC 7452 32-core CPUs while MI100 nodes are equipped with two AMD EPYC 7552 48-core CPUs. A more detailed description of the hardware is given in Section 2.1.2. The software environment consisted of ALMA Linux 8.7 as the operating system, used across the EPN farm.

To conduct the tests, a simulated timeframe was used consisting of Pb–Pb collisions at 50 kHz interaction rate spanning 128 LHC orbits, generated via Pythia 8.3 [13] with the ALICE configuration for Pb–Pb interactions. This dataset was chosen to represent a realistic workload encountered during data taking at high-intensity Pb–Pb collisions.

3.2 TPC Cluster Finder Summary

This section describes the basic functionality of the TPC cluster finder. These steps were originally developed in the author's master thesis [74], which contains a more thorough discussion on the implementation and functionality of the cluster finder.

The cluster finder combines charges neighboring in pad (space) and time direction into clusters and consists of four steps. In a first step, local maxima called peaks are found. The second step applies a noise filter to the found peaks, discarding false positives. Afterwards, charges are split between neighboring peaks. Finally, charges around all peaks are combined into clusters. Clusters are limited to a single TPC row. No clusters can be found across rows. All rows of a single TPC sector are processed at once.

The input to the clusterizer is a two-dimensional array (with a pad and time axis) that contains all charges within the timeframe or subtimeframe respectively, this array is dubbed `chargeMap`. The second input parameter is a list of all positions with non-zero values on the `chargeMap`. The individual steps of the clusterizer process the positions on this list in parallel, or on a list of positions produced by a previous step.

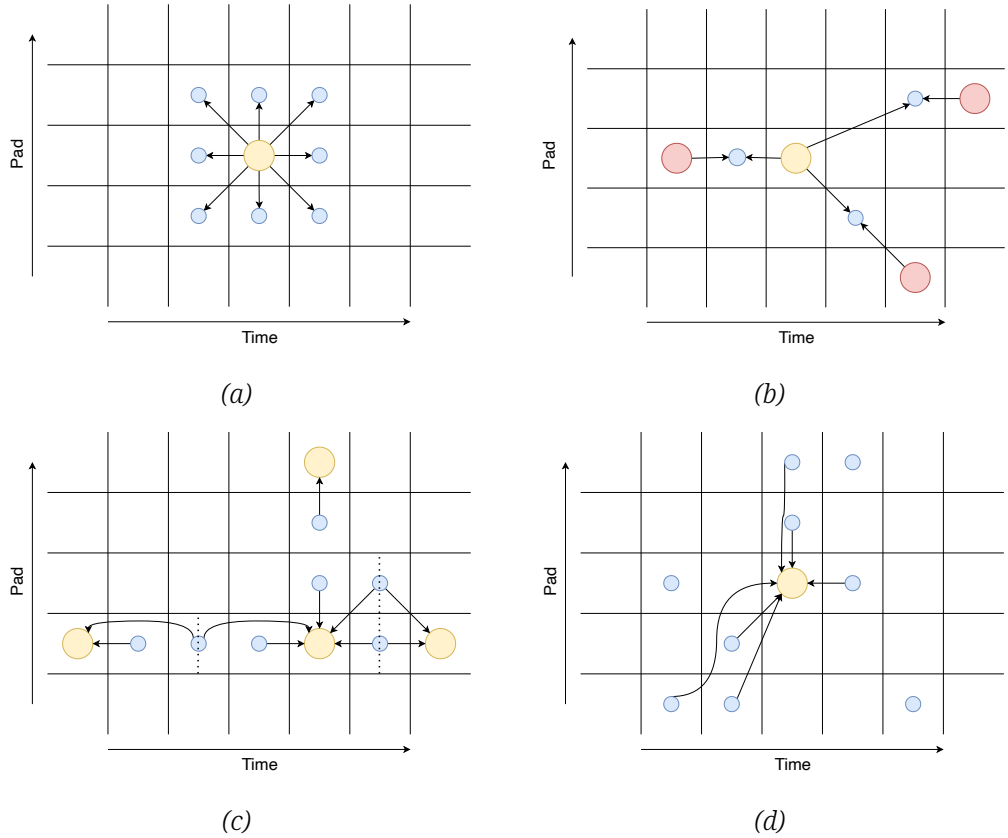

Figure 3.1: The different substeps of the cluster finder. Figure (a) illustrates the peak finding step. Figure (b) shows the following noise suppression. Figure (c) shows the deconvolution step, that splits charges between neighboring clusters. Lastly, Figure (d) shows how neighboring charges are combined into a cluster. Note that charges in the 5×5 neighborhood are only added if they have a corresponding charge in the 3×3 neighborhood. Thus three of the outer charges are not part of the cluster in this example.

Figure 3.1a outlines the peak finding. Each ADC value is compared with it's eight immediate neighbors. Values that are local maxima in this 3×3 area are marked as peaks.

The noise suppression is shown in figure 3.1b. Each peak is compared with all other peaks in a 7×5 area. For each pair the smaller one is discarded, if there is no minima between both peaks. A minima is this case is defined as $|p - q| < \varepsilon$, where p is the ADC value of the smaller peak, q is the ADC value of the potential minima and ε is a configurable parameter.

During the deconvolution step, ADC values are split between their neighboring peaks. The process is shown in figure 3.1c. For each ADC value the number of peaks in 3×3 and 5×5 neighborhoods are counted. The charge is then divided by the number of peaks in the 3×3 are, if there are any. Otherwise, it is divided among the peaks in the 5×5 area.

After the deconvolution was applied to all charges, a cluster is created around every peak as shown in figure 3.1d. ADC values in the 3×3 neighborhood are always part of the cluster. Additionally, charges in the 5×5 area around the peak are added to the cluster, if they have no peak in their 3×3 neighborhood.

The complete TPC reconstruction chain consists of multiple components that support the core cluster finding algorithm. As shown in Figure 3.2, two processing paths exist - one for real detector data and one for simulated data. For real data, the processing begins with zero suppression decoding of the raw data stored in 8 kB pages, followed by filtering of noisy pads that could interfere with cluster finding. Simulated data bypasses these steps, instead requiring only the filling of the charge map from the generated digits. Both paths then converge at the cluster finder, which follows the steps outlined previously. When processing simulated data, an additional re-run of the clusterizer kernel on the CPU propagates the Monte Carlo truth information by tracking which simulated particle contributed to each cluster. This allows validation of the GPU-based reconstruction chain by comparing its output against the known true particle trajectories that generated the signals.

3.3 Dividing Timeframes

The TPC clusterizer must handle timeframes that can span up to 100 000 time bins. At this scale, the memory required by the clusterizer can exceed the available GPU memory. To address this limitation, the clusterizer implements a fragmentation strategy that divides timeframes into smaller subtimeframes while ensuring consistent cluster finding results.

The core of this approach is the `CfFragment` class, which represents a contiguous section of time bins within the larger timeframe. Each fragment maintains an overlapping region of 8 time bins with its neighboring fragments, ensuring that clusters

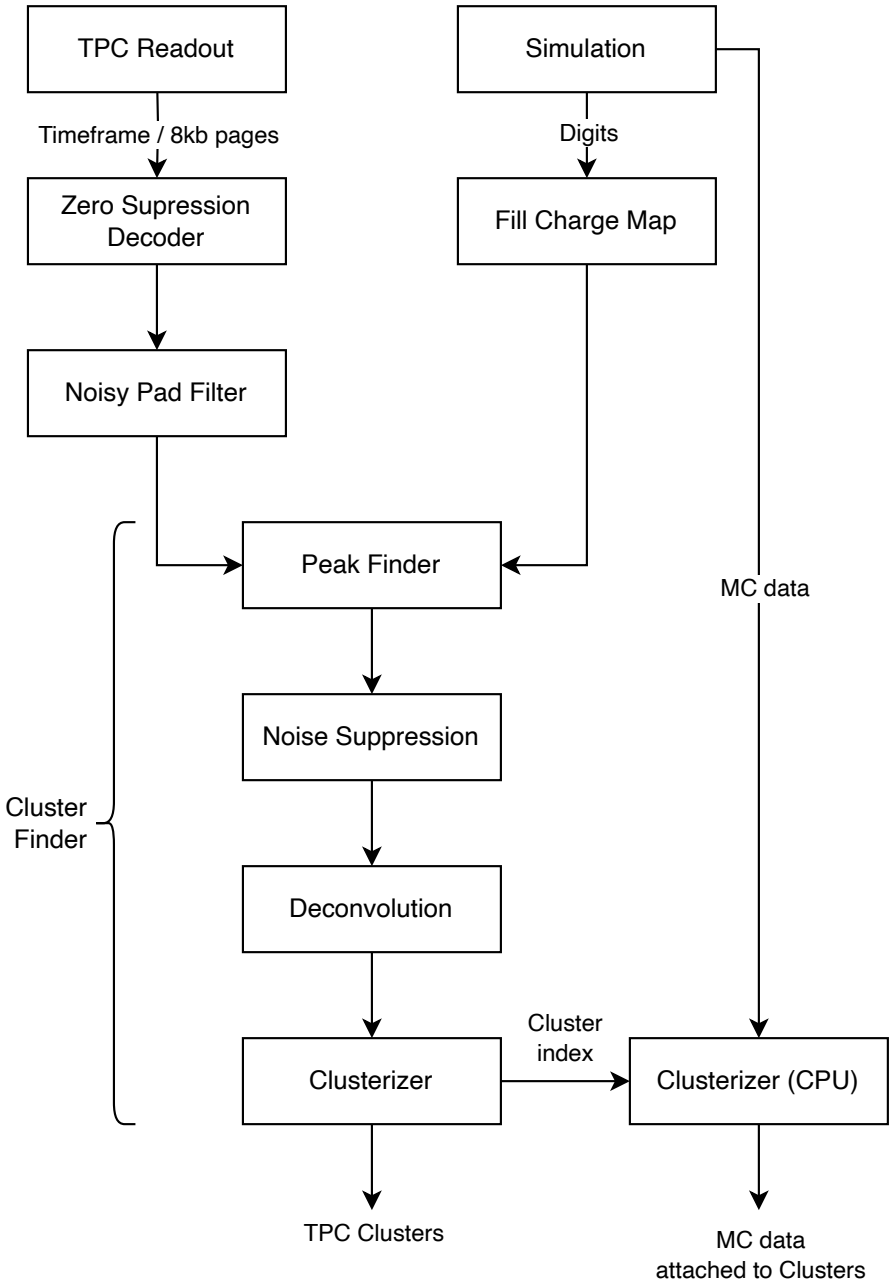

Figure 3.2: The different steps of the cluster finder with simulated and real data.

spanning fragment boundaries are properly reconstructed. This overlap is particularly crucial because the cluster finder examines a 5×5 window around potential peaks, which could otherwise be split across fragment boundaries.

The clusterizer processes fragments sequentially, with each fragment handling 4000 time bins by default. This allows parallel processing of three TPC sectors, enabling overlapping of computation and data transfer. For a fragment with index i, the time range processed is:

$$[t_{\text{start}} + i \cdot (l - 2o), t_{\text{start}} + (i + 1) \cdot l]$$

where l is the fragment length and o is the overlap size (8 time bins). The $2o$ term in the stride accounts for the double counting of overlap regions - each internal time bin appears exactly once in the final output, while bins in overlap regions are processed twice but only counted in one fragment.

When processing overlap regions, the clusterizer must carefully handle cluster assignment to ensure consistent results. For each fragment after the first, the first 8 time bins are treated as backlog, and their clusters are discarded since they were already processed in the previous fragment. Similarly, for fragments before the last, the final 8 time bins are marked as future overlap, and their clusters are retained to ensure continuity with the next fragment.

This fragmentation strategy effectively enables processing of arbitrarily large timeframes with limited GPU memory while maintaining the quality of cluster finding. The overlap mechanism ensures that no clusters are lost or malformed at fragment boundaries, producing results identical to processing the entire timeframe at once.

3.4 Zero Suppression Decoding

The TPC data is transmitted using zero-suppression (ZS) encoding to reduce bandwidth requirements. In this format, only channels that recorded charge depositions above a threshold are stored, organized into 8 kB pages. The task of decoding this data efficiently on GPUs presents several challenges, particularly in terms of memory access patterns and parallel processing.

Over time, the TPC zero-suppression format has evolved to better handle the increasing data rates in Run 3. The initial row-based format was replaced by a link-based version in 2022, which was subsequently refined into the dense-link-based format to achieve better compression ratios necessary for handling high-intensity Pb–Pb collisions. In this section the implementation of the GPU decoder of the two link-based formats for online processing is discussed.

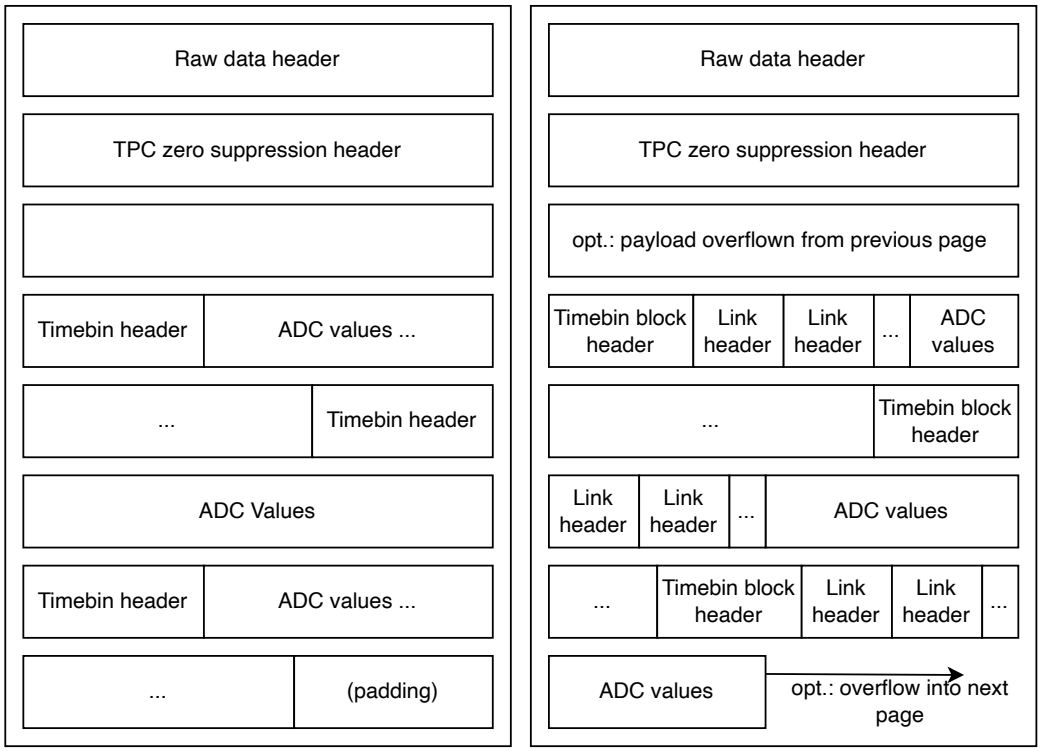

Figure 3.3: Approximate page layout of link based and dense-link based zero suppression formats, shown respectively in Figure (a) and Figure (b).

```
int warpPredicateScan(int pred, int* sum) {
  int iLane = hipThreadIdx_x % warpSize;
  uint64_t waveMask = __ballot(pred);
  uint64_t lowerWarpMask = (1ull << iLane) - 1ull;
  int myOffset = __popcll(waveMask & lowerWarpMask);
  *sum = __popcll(waveMask);
  return myOffset;
}
```

Figure 3.4: Optimized variation of the warp scan operation in HIP for computing the prefix sum over boolean values. The register shuffles and additions are replaced with a single call to the `ballot` intrinsic. The prefix sum is then given by a popcount of the n least significant bits, where n is the thread position within the warp. This version additionally computes total sum with an additional popcount call as this value is needed by zero suppression decoding. The implementation in CUDA is equivalent but uses 32 bit integers to store the bitmasks instead due to the smaller warp size.

Before decoding can begin, pages must be assigned to the appropriate fragment of the timeframe (compare Section 3.3). This is accomplished by examining the time information in each page's header and computing which pages contribute to each fragment. As pages may contain data spanning fragment boundaries, this must account for potential overlap. Pages that fall between two fragments are necessarily decoded twice and ADC values that fall outside of the current fragment are discarded after decoding.

Each page begins with a standard raw data header containing metadata such as the CRU (Common Readout Unit) identifier and heartbeat orbit information. This is followed by TPC-specific headers and the actual zero-suppressed ADC values. The decoding process must extract these values and place them into a three-dimensional charge map indexed by pad, row, and time required by the clusterizer.

Regardless of the format, each decoded ADC value requires additional memory accesses: one for gain correction from a calibration table, and in the case of link-based formats, another access to map the front-end card position to global pad coordinates. These scattered memory accesses can impact performance, particularly on architectures with limited cache capacity.

3.4.1 Link Based Zero Suppression

The layout of a link-based zero suppression page is outlined in Figure 3.3a. Every 8 kB page begins with a format-independent raw data header. This header contains, among other things, information on the CRU where the page originated from and the heart beat frame during which the data was collected. The next 128 bytes consist of the TPC ZS header containing metadata about the page, like the number of timebins and offset relative to the first timebin.

Each timebin contains the data of up to 80 neighboring channels collected during a single tick. These timebins begin with another 128-byte header where the first 80 bytes form a bitmask indicating which channels have data. The header is then followed by the ADC values for those channels. Within a page, timebins are stored sequentially with padding added at the end of a page if needed to maintain the 8 kB page size.

The decoding is parallelized at two levels: across pages at the block level and across channels within each page at the thread level. The decoder first performs a parallel scan over the channel bitmask to determine the output positions for each channel's data. This scan operation is implemented efficiently using warp-level primitives to minimize synchronization overhead, as shown in Figure 3.4.

This optimized implementation replaces multiple register shuffles and additions typically needed for scan operations with ballot and population count intrinsics. The ballot operation creates a bitmask where each bit represents whether a thread in the warp has data to process. A population count on the bits below a thread's position

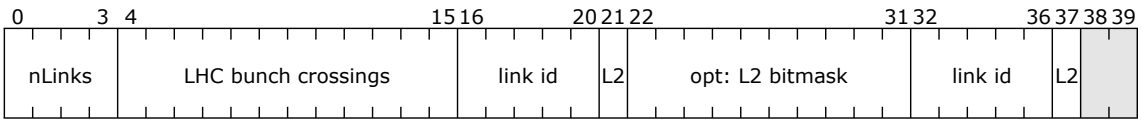

Figure 3.5: Layout of a timebin block header in the dense link-based zero suppression format.

then gives its offset in the output, while a population count of the full mask provides the total number of active channels.

Each GPU thread is assigned a channel within the timebin. For channels marked as active in the bitmask, the thread decodes the corresponding ADC value and writes it to the charge map at the position determined by the scan operation. The thread also applies the gain correction and maps the front-end electronics position to global pad coordinates. This mapping requires additional memory lookups but is necessary to convert the raw data to the detector geometry needed for reconstruction.

3.4.2 Dense-Link Based Zero Suppression

The dense-link-based ZS format introduces several optimizations to achieve higher compression rates necessary for high-luminosity Pb–Pb data processing. As shown in Figure 3.3b, the format follows a similar overall structure with raw data and TPC-specific headers, but introduces key changes to the timebin organization. Instead of processing each link separately, multiple links are collected into timebin blocks, and the last block in a page can overflow into the next page, eliminating padding waste.

Each timebin block begins with a compact header, detailed in Figure 3.5. The first 4 bits are the number of links contained within the timebin block. The next 12 bits are the LHC bunch crossings within the timeframe used to compute the TPC timebin of the block. The timebin block header is followed by *nLinks* timebin link headers consisting of the link id and bitmask of active channels, each with a variable size of 3 to 11 bytes. The first 5 bits are the link id within the CRU. The next bit $L2$ indicates if the channel mask has two levels. If $L2$ is not set, the next 10 bits indicate which bytes of the channel mask are present to eliminate zero bytes from the mask. Otherwise all 10 bytes of the mask are present and the two bits after $L2$ are unused.

The dense decoder employs a two-phase approach for each timebin block. In the first phase, link headers are processed sequentially to determine data offsets and construct an index of channel positions. This phase cannot be fully parallelized due to the variable-sized headers. The second phase decodes ADC values with full warp utilization, with each thread processing a single ADC value based on the indices computed in the first phase.

Special handling is required for timebin blocks that overflow into the next page. The implementation provides two code paths: an optimized version for standard blocks

that omits bounds checking and a more conservative version that performs bounds checking when crossing page boundaries. The latter implementation is necessary, as the overflow can happen at an arbitrary point in the timebin block. The appropriate version is selected based on flags in the page header, ensuring maximum performance for the common case while maintaining correctness for overflow blocks.

To minimize thread divergence, the implementation first uses a warp-level parallel scan to determine positions for ADC decoding, similar to the approach used in the link-based decoder. However, the scan must now account for multiple links within the same timebin block. The decoder maintains a shared memory buffer containing channel indices and link identifiers, allowing threads to efficiently look up their target positions in the output charge map while minimizing global memory accesses. [1]

Cache utilization is improved by processing ADC values in order within each timebin block, maximizing spatial locality for the calibration table lookups. The implementation also ensures coalesced memory accesses when writing decoded values to the charge map by maintaining proper alignment of output positions within each warp.

3.4.3 Performance

Parallelism on CPU

To enable fair comparison between CPU and GPU performance, careful consideration must be given to the CPU parallelization strategy. The O^2 framework employs a two-level parallelization approach: the outer level distributes work across TPC sectors, while the inner level processes the 8 kB pages within each sector in parallel. This creates a challenge in thread allocation, as the 36 TPC sectors suggest the outer thread count should be a factor of 36, which conflicts with the 128 hardware threads available on the test system (compare Section 3.1.1).

For N total threads, the O^2 framework allocates $\lceil \frac{N}{36} \rceil$ threads to the inner loop while maintaining one thread per sector for the outer loop. Figure 3.6 illustrates the performance scaling of the dense link-based decoder with increasing thread counts. A notable observation is the significant performance jump when moving from 64 to 72 threads, where the speedup factor increases from 29 to 47. This occurs because both configurations process TPC sectors with 2 threads, but with 64 total threads, the OpenMP [30] runtime cannot execute all sectors simultaneously, leading to processing stalls.

The impact of simultaneous multithreading (SMT) is particularly pronounced in this workload. Using all 128 virtual cores (64 physical cores with SMT) achieves a speedup

[1] On the AMD MI50, 1.6 kB of shared memory can be allocated per warp before occupancy is reduced. For reference, each compute unit has 64 kB and can run up to 40 warps at once. Storing the number of samples per timebin block, the link ids and the channel index per ADC requires $2*16+16+80*16 = 1328$ B. Thus the allocated shared memory doesn't reduce occupancy for the decode kernel.

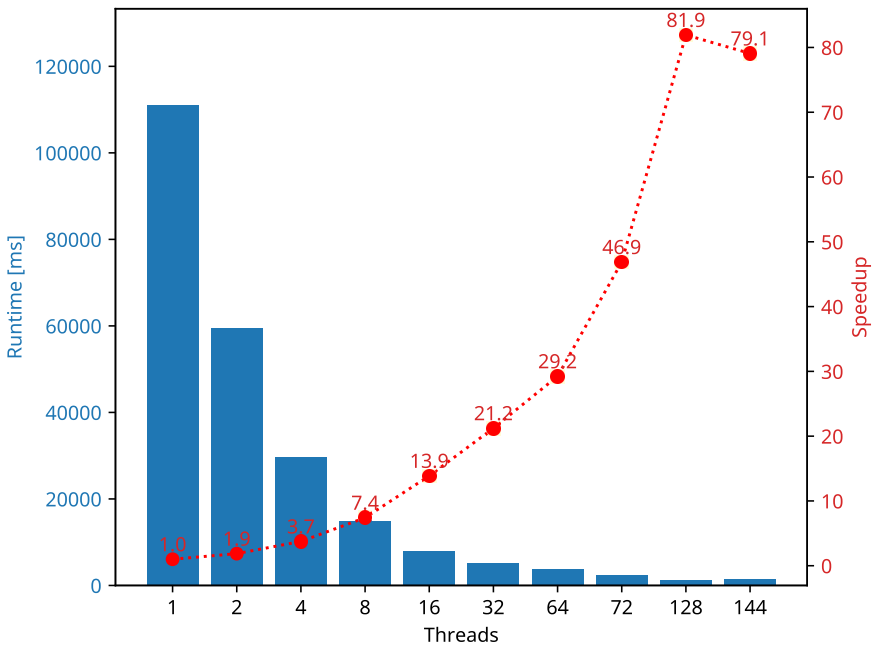

Figure 3.6: Speedup of the dense link-based decoder without thread affinity.

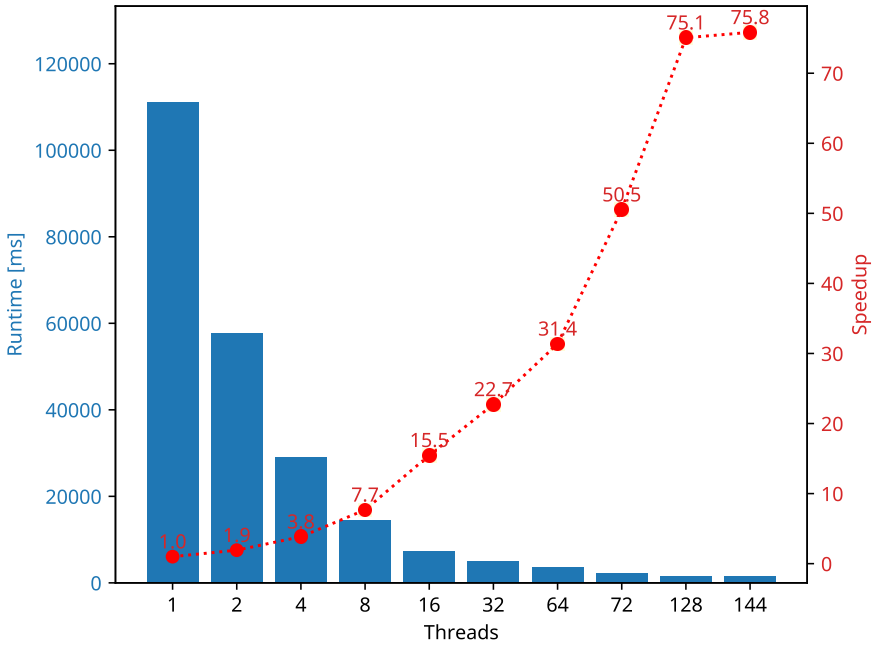

Figure 3.7: Speedup of the dense link-based decoder with threads pinned to NUMA domains.

of 81, significantly outperforming both the 72-thread case and the 144-thread oversubscription scenario. As the decoder is highly memory bound, a possible explanation for the improvement with SMT is that the CPU can hide latency by switching between hardware threads when a stall, due to memory access, occurs.

An intuitive optimization approach would be to pin threads processing the same TPC sector to a single NUMA domain, theoretically eliminating cross-domain memory access overhead. This was implemented by setting

`OMP_PLACES=sockets` and `OMP_PROC_BIND=spread,master`

to pin first-level threads to NUMA domains while keeping second-level threads within the same domain. As shown in Figure 3.7, this strategy provides modest performance improvements at lower thread counts. However, when the CPU is fully utilized with 128 or 144 threads, performance actually degrades compared to the unpinned case, with the speedup factor dropping from 81 to 75. This suggests that the operating system's scheduler can better balance resources when given the flexibility to move threads between domains.

A more granular approach, discussed in detail in Appendix B.3, attempts to maximize memory bandwidth by distributing TPC sectors across Core Complex Dies (CCDs), each with its own L3 cache. While this further improves performance at lower thread counts, it similarly suffers from reduced efficiency under full CPU utilization like the socket-level pinning.

The performance characteristics described above were measured on the AMD EPYC 7452 CPU, which comprises the majority of the processing farm's MI50 nodes. While the MI100 nodes use the EPYC 7552 with a higher core count, it exhibits the same single-thread performance and scaling behavior, as demonstrated in Appendix B. The appendix also includes comprehensive performance data for all three decoding algorithms, which show consistent scaling patterns despite differences in absolute performance and maximum achievable speedup.

GPU Performance

Figure 3.8 shows the GPU runtime performance for the three zero suppression formats across different processing architectures. The performance characteristics exhibit interesting patterns when moving from CPU to GPU execution. While the row-based format performs best on CPU, it shows the worst performance on GPUs, requiring around 1155 ms on the MI50. The link-based format achieves intermediate performance across all platforms, with runtimes of approximately 1246 ms and 823 ms on the MI50 and MI100, respectively.

The dense link-based format, despite having the poorest performance on CPU, achieves the best runtime on both GPU architectures. It processes timeframes in around 900 ms

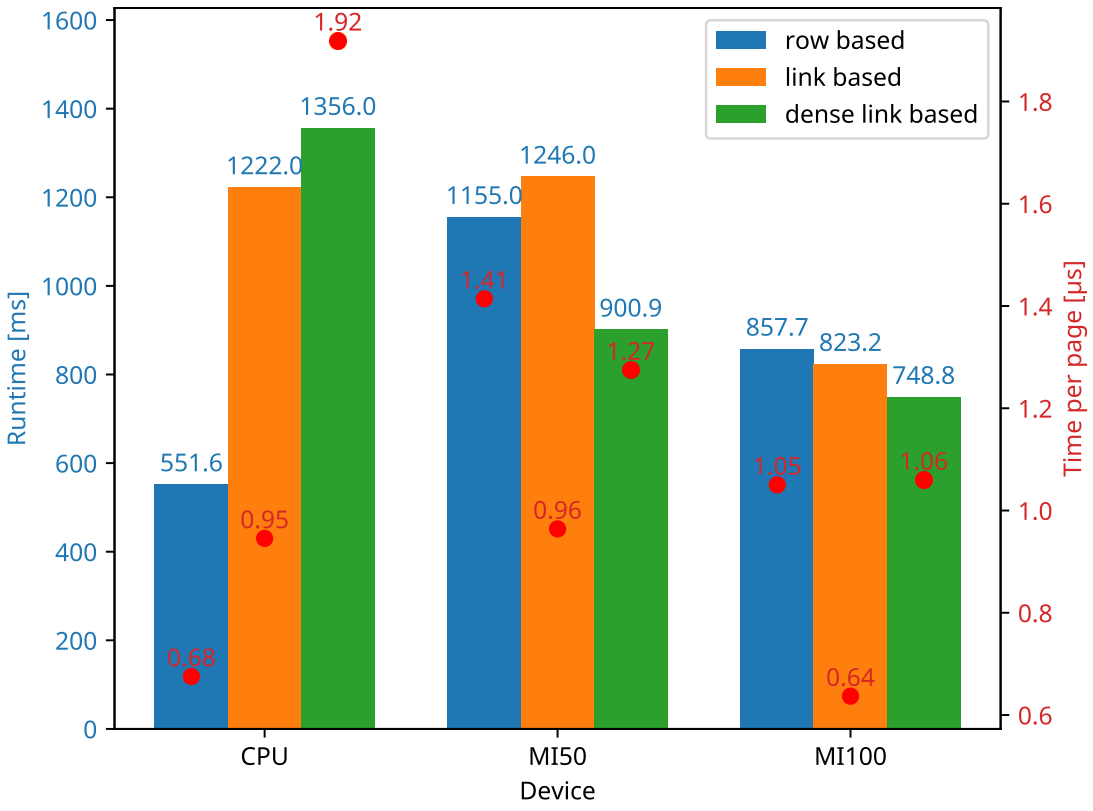

Figure 3.8: Runtime comparison of the different Zero Suppression formats on the MI50, the MI100 and running on CPU. The vertical lines show the total time spent to decode the entire timeframe. While the average times spent per page are represented by the red markers.

on the MI50 and 750 ms on the MI100, with the MI100 consistently delivering 20 % to 30 % better performance across all formats. However, when examining the time spent per page, an interesting pattern emerges. The dense link format exhibits the highest per-page processing time on CPU at 1.92 µs, and while it improves to 1.0 µs to 1.2 µs on GPU, this is still relatively high compared to the other formats.

This higher per-page cost stems from the format's primary optimization goal - maximizing compression ratio to support higher network throughput rather than optimizing for GPU processing patterns. The format achieves better overall performance on GPUs despite higher per-page costs simply because it processes fewer pages in total.

3.5 Noisy Pad Filter

The Noisy Pad Filter is responsible for detecting and excluding TPC pads that have lost their baseline during data taking. Such pads can emit a continuous stream of noise that would interfere with cluster finding. To efficiently detect these pads, the filter analyzes charge patterns by examining two key metrics for each pad: the total number of charges above threshold and the maximum number of consecutive time bins containing charges. The filter processes TPC data in parallel by assigning multiple pads to each thread block. For efficient memory access, pads are processed in groups matching the cache line size. The implementation uses shared memory as a temporary buffer when running on GPUs, allowing threads within a block to cooperatively analyze charge patterns. For CPU execution, the implementation takes advantage of vectorization through the Vc library [47] when available. This enables efficient SIMD processing of multiple pads simultaneously. The vectorized code path processes charges in blocks of eight pads, matching common SIMD widths on modern processors. A pad is marked as noisy if either of two conditions is met:

1. The total number of charges exceeds a threshold calculated based on the timeslice length.
2. The number of consecutive time bins with charges exceeds a configurable threshold.

Additionally, a saturation check is performed - if a pad's maximum charge exceeds a configurable threshold, it is not marked as noisy regardless of the charge patterns. This prevents accidentally excluding pads with valid high-energy deposits. The filter's runtime remains constant regardless of the timeslice length since it processes a fixed number of pads. On the AMD MI50 GPU, the filter typically requires about 35 milliseconds to process a complete timeslice, while the more powerful MI100 completes the same task in approximately 25 milliseconds. This represents only about 0.1 % of the total timeslice processing time, making it a relatively inexpensive step.

3.6 Monte Carlo Propagation

To evaluate the results of reconstruction algorithms, data generated by simulations also contains ground truth data, called Monte Carlo labels. These map digits onto the particle tracks that contributed to the digit. Labels are propagated along the reconstruction chain. The labels of TPC clusters are the set of labels from all digits that contributed to the cluster.

Extending the cluster finder to support Monte Carlo (MC) labels poses several problems. The number of labels a cluster will have is not known beforehand. In principle all tracks could contribute to the same cluster, but this is not a useful upper bound. This makes static memory allocation beforehand for labels difficult. Therefore collecting labels purely on GPU is very complex and dynamic allocation per cluster instead is required. Furthermore, the compute overhead from MC labels must be minimized for online-scenarios where no labels are present.

Monte Carlo labels are stored as integers that represent the corresponding track id. Input to the cluster finder in addition to an array of digits is an array containing the labels with a second array that maps digits to the offset where it's labels are stored. The output of the cluster finder has the same structure. An array of clusters with an array of labels and a mapping from clusters to the offset in the label array.

Labels are collected for every cluster during cluster creation. This is very natural, as all digits that belong to the cluster are iterated at this point anyway. As the label collection requires dynamic memory allocation it is removed at compile time from GPU code. This removes any overhead for online processing and allows the usage of STL containers. In the initial implementation `std::unordered_set` was used for collection to prune duplicate labels. However this turned out to be too slow and was replaced by a `std::vector` with a bloom filter. Runtime of that kernel improved by a factor 5 with this change.

Combining the collected labels into a flat array is done with a simple reduction. The number of labels in each TPC row can be computed in parallel. With this we have the total number of collected labels to allocate the output memory. Now the offset of every row can be calculated and the labels are copied to the corresponding position in the target array. This step is trivial parallel across TPC rows.

To fully evaluate GPU reconstruction, using only CPU labels isn't sufficient. While in principal both versions should produce identical clusters, differences might still be present due to hardware differences in handling of floating-point numbers or software bugs. When labels are present, the cluster id is stored alongside the accompanying peak on GPU. All relevant buffers are then transferred back to the host and the clusterizer kernel is run again on CPU to collect only labels and discard the clusters.

3.7 Parallel Track Merging

Given a list of track slices and links of which slices should be connected to tracks, the track merger kernel finds the positions where slices should be connected. This is not a very expensive kernel, but it is too slow when run fully serial with only a single GPU thread. This is hard to parallelize without locking. One would need to check for collisions when threads are writing to the same slice and potentially add a repair step.

Instead of attempting to parallelize the track merging itself, one could instead look for independent set of track slices. Between these sets no collisions are possible and can be processed in parallel. Track slices and links can be interpreted as vertices and edges of a graph. The connected components of this graph are the independent sets we look for. Finding the connected components in a graph is a basic problem in computer science for which efficient (and surprisingly simple) GPU algorithms already exist [67]. Each GPU block can then process separate connected components to merge track slices.

3.7.1 Perfomance

To allow a fair comparison between the sequential and parallel implementations, the sequential track merger was backported into the current O^2 version. This ensures both versions operate on the same data structures and time frames.

Figure 3.9 shows the performance comparison between the sequential and parallel variants on both the AMD MI50 and MI100 GPUs, with timings including the overhead from the connected component detection. The parallel implementation achieves remarkable speedups - over 30x on the MI50 (from 6739 ms to 204.9 ms) and 47x on the MI100 (from 7607 ms to 159 ms). It's worth noting that while the connected components algorithm introduces some overhead, its contribution to the total runtime is minimal - less than 5 % of the execution time.

3.8 Cluster Gathering

After track reconstruction, TPC clusters need to be copied from GPU memory back to the host. The clusters are stored in a Structure of Arrays (SoA) format and are scattered in memory according to their track assignments. A naive approach to copying these clusters would require millions of small memory transfer operations, which would be highly inefficient. Several strategies have been implemented to handle this data movement efficiently.

The initial approach uses Direct Memory Access (DMA) to write directly to pinned host memory. To minimize the impact on concurrent processing, this kernel uses only 1–2 blocks, each occupying a single compute unit, allowing the data transfer to run in

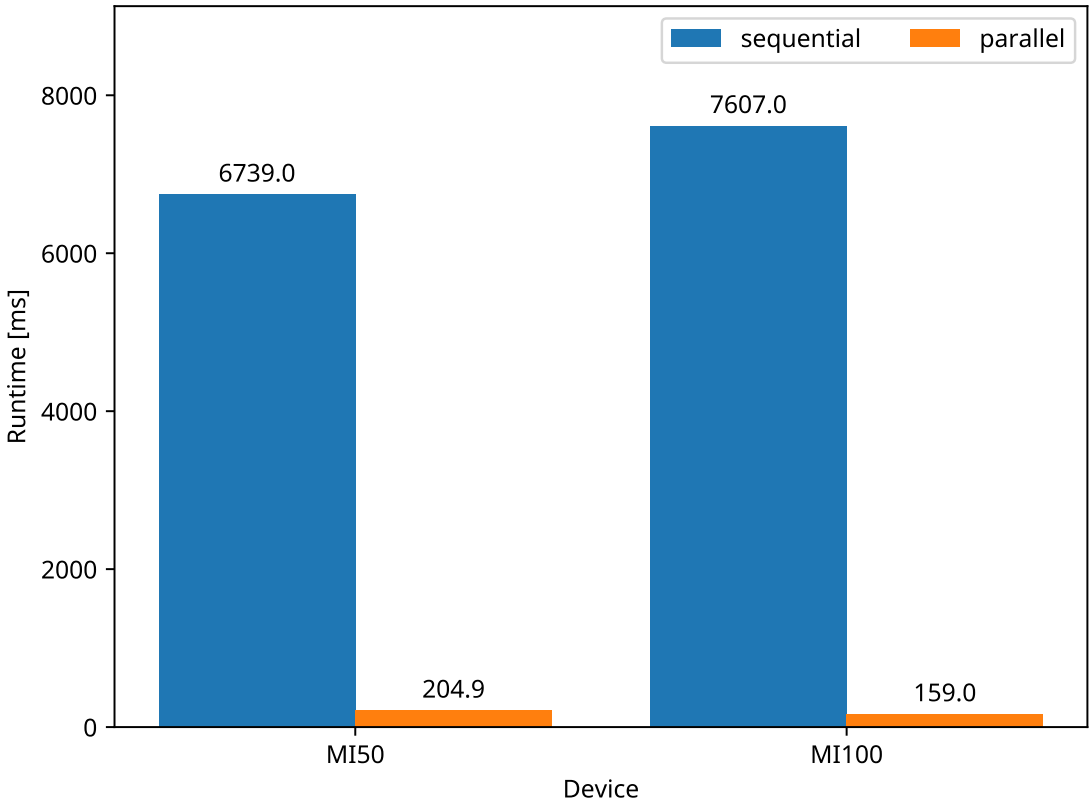

Figure 3.9: Perfomance of the Track Merger kernel before and after parallelizing over the connected components.

the background while the next timeframe is being processed. Each block processes clusters sequentially, writing them directly to host memory.

An improved version maintains the same execution strategy of using only 2 compute units but introduces shared memory buffering to optimize memory transactions. The kernel first performs a small write to align the destination pointer in host memory, then accumulates clusters in shared memory buffers. Once a buffer is full, it is written to host memory in a single operation with writes of 128 bytes[2]. This buffering strategy helps coalesce memory writes and reduces the total number of DMA operations.

The current implementation used in ALICE online processing, the `multiBlock` kernel, takes a different approach by utilizing the full GPU device. Instead of writing directly to host memory, this kernel first compacts the scattered cluster data into contiguous device memory. The work is divided between cluster types: attached clusters (belonging to tracks) and unattached clusters. Odd-numbered blocks handle unattached clusters while even-numbered blocks process attached clusters, enabling better load balancing.

The kernel uses a warp-based approach for data movement, where each warp handles a contiguous section of clusters. The work distribution is calculated using parallel prefix sums (scans) to determine the output positions for each warp. When processing attached clusters, the implementation uses a cooperative approach where threads within a warp work together to process clusters from the same track, ensuring efficient memory access patterns despite the variable-length nature of the data.

After the clusters are compacted in device memory, a single large memory transfer operation copies the data back to the host. Only this final transfer operation overlaps with the processing of the next timeframe. This approach effectively balances the competing goals of efficient data movement and minimal impact on concurrent processing. By performing the compaction on the GPU and reducing the host transfer to a single operation, the implementation achieves better overall system throughput compared to the earlier approaches.

3.8.1 Performance

Figure 3.10 compares the performance of four different approaches to cluster gathering, tested on both the AMD MI50 and MI100 GPUs. The naive approach of performing individual DMA transfers to host memory for each track proves prohibitively slow, requiring approximately 200 s per timeframe on both devices. This poor performance is likely bottlenecked by driver overhead from the numerous small memory transfers rather than hardware limitations, as evidenced by the similar execution times across different GPU architectures.

[2]This is done via `uint4`, the biggest vector type supported by the GCN 5 architecture.

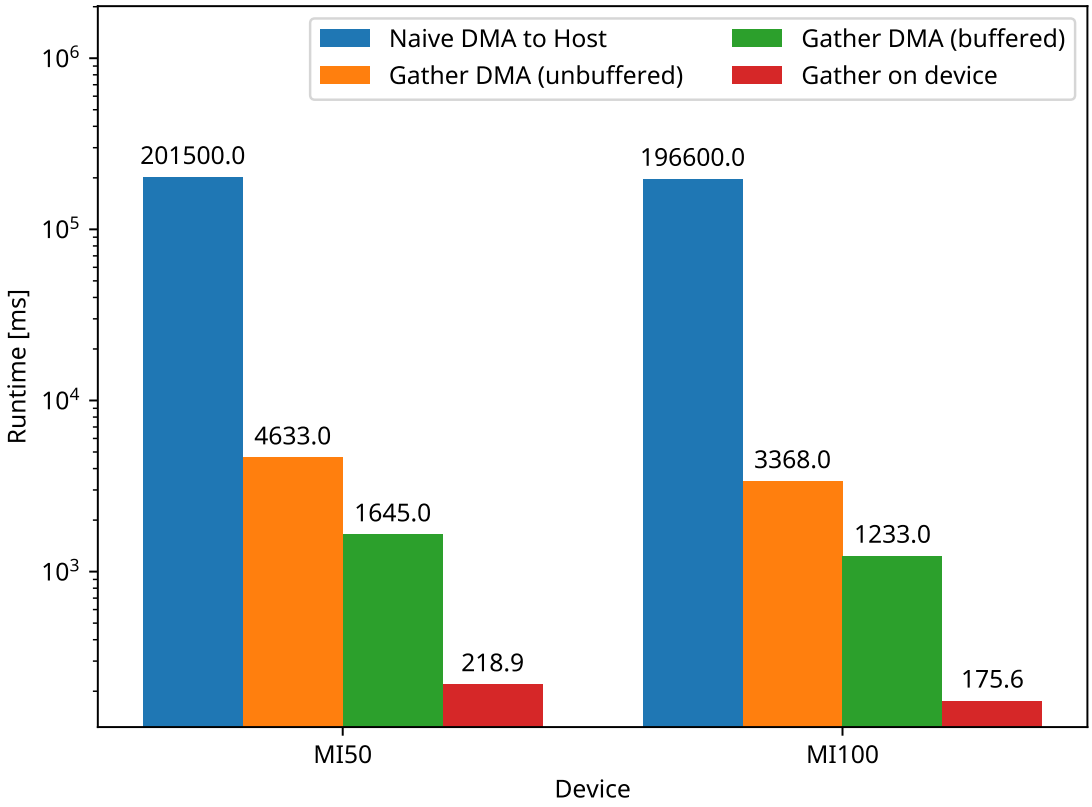

Figure 3.10: Performance of different gather algorithms on the MI50 and MI100.

The gather kernel writing directly to host memory via DMA, shown in orange, shows a dramatic improvement, executing 43 times faster on the MI50 and nearly 60 times faster on the MI100 compared to the naive approach, shown in the blue bar. Adding shared memory buffering to coalesce memory writes further improves performance by roughly a factor of 3 on both devices. The buffered implementation achieves this by accumulating clusters in shared memory and performing fewer, larger DMA transfers.

Finally, performing the gather operation entirely on the device using all available compute units before a single transfer to host memory yields another significant speedup of approximately a factor of 7. This approach proves most efficient by maximizing GPU utilization and minimizing PCIe transfers.

Figure 3.11 provides a detailed breakdown of the execution time for this final approach. On the MI50, about one quarter (60.4 ms) of the processing time is spent in the gather kernel, with the remaining time (158.5 ms) taken up by the DMA transfer back to host memory. The ratio is even more pronounced on the MI100, where only about one fifth (38.4 ms) is spent gathering, leaving the majority (137.2 ms) for data transfer.

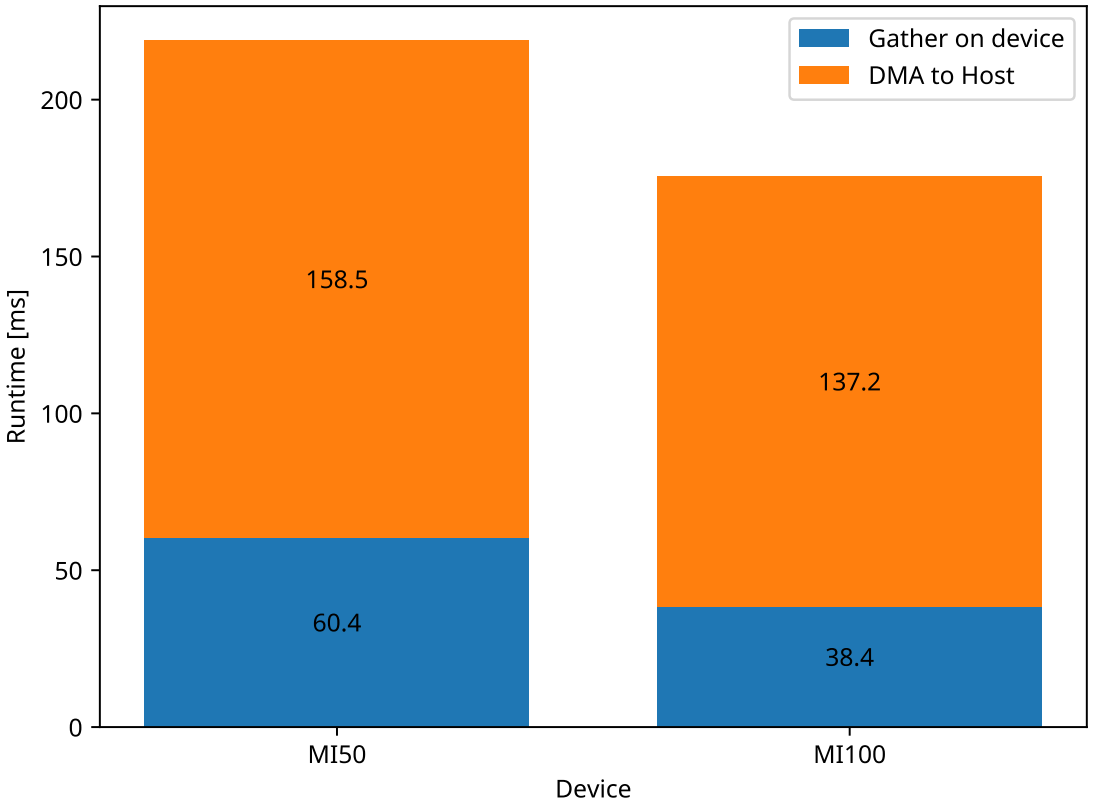

Figure 3.11: Breakdown of cluster gathering times for gathering on device memory and performing DMA transfer.

However, since this final DMA transfer can overlap with the processing of the next timeframe, the actual performance impact is minimal. The data suggests that PCIe bandwidth remains the fundamental bottleneck for cluster gathering, as evidenced by the substantial portion of time spent in data transfer across both devices.

Chapter 4

xpu - A C++ Libary for Portable GPU Code

4.1 Introduction

Modern high-energy physics experiments increasingly rely on GPU computing to handle their demanding data processing requirements. (See Chapter 1 for examples) However, developing GPU-accelerated software for these experiments presents unique challenges: code must be portable across different GPU architectures, maintainable over long time periods, and capable of achieving high performance without sacrificing flexibility. Moreover, the code often needs to run efficiently on CPUs for development and testing, while still integrating with existing C++ codebases.

In the course of this thesis, xpu was developed as a lightweight C++ library to address these challenges. A particular focus were the needs of the CBM experiment's online processing system (compare Chapter 5) that shaped the development and in turn served as a testbed for the library. It provides a unified interface for CUDA, HIP, and SYCL while allowing code to run on CPUs, enabling developers to write portable GPU code without sacrificing performance or control over hardware-specific optimizations. The library's design emphasizes explicit memory allocations, separate compilation of device code, and zero-overhead abstractions - features particularly important for high-energy physics applications where performance and predictability are crucial.

At its core, xpu takes a pragmatic approach to GPU abstraction. Rather than attempting to hide hardware differences completely, it provides a thin layer that maps efficiently to native GPU programming models while maintaining consistent semantics across platforms. This approach allows developers to reason about performance characteristics and utilize platform-specific features when needed, while still writing portable code.

This chapter describes the design and implementation of xpu. It structured as follows:

- The following two Sections 4.1.1 and 4.1.2 explain the motivation and requirements that shaped the development and discuss existing alternatives and their limitations and shortcomings that xpu attempts to address.
- Section 4.2 describes the chosen architecture and how xpu achieves portability through separate compilation and dynamic loading of device code.
- The following Section 4.3 details the library's key features including memory management, kernel execution, and profiling capabilities.
- Section 4.4 goes into detail how the dynamic loading of device code is handled.
- The chapter concludes in Section 4.5 with a performance analysis comparing xpu to native implementations and alternative abstraction approaches.

4.1.1 Motivation

The development of xpu was driven by the specific requirements of high-energy physics (HEP) experiments, particularly the needs of the CBM experiment's online processing system (compare Chapter 5). With data taking scheduled to begin in 2028, CBM faces unique challenges in its computing infrastructure that existing GPU programming solutions do not fully address.

A primary consideration is hardware flexibility. While current planning envisions GPU-accelerated processing in the Green IT Cube computing center, the specific hardware that will be available in 2028 cannot be determined with certainty. This necessitates a programming model that can target multiple GPU architectures without requiring significant code changes. Additionally, the ability to run the same code on CPUs is crucial for development, debugging, and providing a fallback option for processing pipelines.

Beyond hardware portability, several key requirements emerged during development:

1. Performance Transparency
   The abstraction layer must not introduce significant overhead in device code compared to native implementations. This is particularly critical for online scenarios in HEP experiments where performance is critical to reduce the amount of hardware needed to process live data. Any abstraction must compile down to essentially the same machine code as direct CUDA or HIP implementations would produce.
2. Explicit Memory Management
   While automatic memory management can simplify development, HEP processing chains require precise control over when and where memory is allocated. This is crucial for:

- Managing limited GPU memory efficiently during continuous data taking.
- Ensuring predictable performance by avoiding hidden allocations.
- Enabling zero-copy operations when beneficial for performance.
- Supporting different memory types (device, host-pinned, managed) depending on access patterns.

3. Compiler Independence
   Different GPU vendors often provide highly optimized compilers for their architectures. For example, NVIDIA's nvcc compiler can produce more efficient code for CUDA than generic solutions. xpu's design allows using the optimal compiler for each target platform while maintaining a unified interface, unlike approaches that require using a single compiler for all backends.
4. Modern C++ Integration
   The library should support modern C++ features and idioms while remaining lightweight and maintainable. This includes:
   - RAII-based resource management
   - Type safety through templates
   - Clear separation between host and device code

These requirements led to the following key design decisions that differentiate xpu from existing solutions:

- Device code is compiled separately for each backend, allowing the use of vendor-specific compilers and optimizations.
- Memory management follows RAII [55] principles while providing explicit control over allocations.
- Platform-specific features (like constant memory) are exposed through a common interface where possible.
- The library maintains a minimal codebase (< 5000 LOC) to ensure maintainability.

This approach provides several advantages for HEP computing environments:

- Development teams can write portable code without sacrificing performance on any platform.
- The same codebase can be used for production GPU processing and CPU-based development / debugging.
- Integration with existing C++ codebases is straightforward.

The following sections detail the technical implementation of these design decisions and demonstrate how they address the requirements of modern HEP computing environments.

4.1.2 Alternatives

Numerous frameworks exist for GPU development, each with their own restrictions and downsides. In this section, I will address some of them in the context of CBM and motivate the development of xpu as an abstraction atop of existing solutions.

The CUDA [49] framework is widely used for GPU programming. However it's restricted to usage with Nvidia GPUs. This makes it unsuitable as the target platform for development in CBM as the code will be only runnable on devices from a single GPU vendor and can't easily be run on CPU.

HIP [8] is an alternative to CUDA developed by AMD. HIP closely follows the CUDA API and enables compilation of device code for AMD GPUs via ROCm [7] and Nvidia GPUs via CUDA. Like CUDA, HIP is primarily a C-API and currently it's not possible to target Intel GPUs. There is also limited support for CPUs via HIP-CPU [72]. However this only enables compilation of the GPU code for the host. Running kernels on the device and host side by side is not possible.

SYCL [40] is a C++ standard for GPU programming. As the successor to OpenCL [65], it also aims to provide a cross-platform API for heterogeneous computing. While SYCL addresses many of the same issues as xpu, some differences and issues remain. The supported platforms depend on the implementation. For example, Intel's oneAPI has a ROCm backend but the supported version can lag behind upstream releases. Furthermore in the SYCL 2020 specification access to constant memory was removed. Lastly the buffer API limits control over when and where memory is allocated. A more detailed discussion on this and the approach chosen in xpu instead is given in section 4.3.2.

Similar to xpu, alpaka [76] is a C++ library aiming to provide an abstraction layer for accelerator programming. However there is a key difference in the approach to managing device code. Alpaka tries to make every platform available that the current compiler supports while xpu compiles code for each platform separately, allowing a different compiler for each platform. The former can simplify some projects as device code doesn't have to be in a separate compilation unit from host code like in xpu. But on the downside this requires the entire device code to be templated to generate different symbols for each backend. Also only one compiler can be used at a time for all backends, whereas xpu can use a different compiler per backend. See sections 4.2 and 4.4 for more detail on how device code is handled in xpu. One notable alternative to Alpaka with similar design goals is the Kokkos library [70].

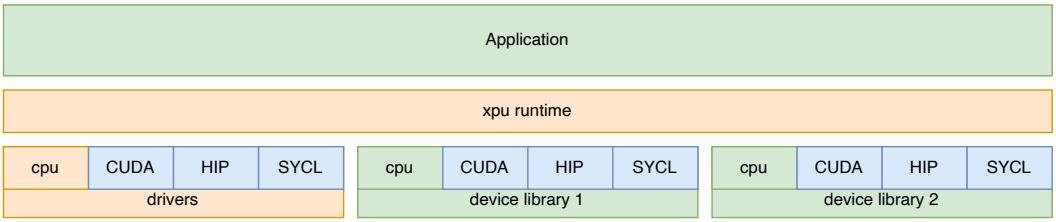

Figure 4.1: Overview of the xpu architecture. Orange color indicates components that are part of xpu. User components are colored in green. Sub components that are loaded at runtime are marked in blue.

4.2 Architecture

The architecture of xpu is designed around three key components working together to enable portable GPU code while maintaining high performance and compiler flexibility. Figure 4.1 illustrates this architecture.

4.2.1 Runtime System

At the core of xpu is a runtime system that manages device discovery, memory allocation, and kernel execution. The runtime provides a unified interface to the application while delegating the actual implementation to backend-specific drivers.

4.2.2 Backend Drivers

For each supported backend (CUDA, HIP, SYCL, CPU), xpu compiles a separate driver as a shared library. These drivers implement the common interface defined by the runtime, providing basic functionality like device discovery and selection, memory operations, and command queue creation. Through this driver architecture, xpu can dynamically load backends based on available hardware and even support multiple backends simultaneously within the same application.

4.2.3 Device Libraries

Device code in xpu is organized into separate libraries that can be compiled independently for each backend.

Each device library undergoes dual compilation: once as regular C++ code for the host, enabling type checking and symbol resolution, and again as device code for each available backend using the appropriate compiler. This approach allows xpu to use the optimal compiler for each platform - for instance, nvcc for CUDA and hipcc for HIP

- while maintaining regular C++ compilation for CPU code without additional dependencies. The result is separately optimized device code for each target platform.

The runtime dynamically loads the appropriate implementation for each kernel based on the active backend through a custom symbol table mechanism. This system maps kernel types to function pointers at during initialisation, providing type-safe access to kernel implementations and enabling efficient switching between backends. This mechanism is discussed in detail in Section 4.4.

4.3 Usage

4.3.1 General Usage

Conceptually, xpu loosely follows the CUDA API. However, as it's written in C++, a more object-oriented approach was chosen and RAII [55] is applied where appropriate. The following paragraphs introduce the basic concepts for using xpu and integration of the library via the build system. Later sections cover different parts of the interface in more detail.

For a full runnable example using xpu, see Appendix A.

The API is divided into three main headers:

- `device.h` - Contains the device-side API for implementing kernels and device functions.
- `host.h` - Provides the host-side API for managing devices, memory, and kernel execution.
- `common.h` - Defines components used by both host and device code.

Additionally, `defines.h` provides essential macros like `XPU_D` and `XPU_H` for marking device and host functions respectively.

Device code in xpu is organized into separate libraries that can be compiled independently for each backend. For implementation reasons, device libraries must define a unique type to identify with and are marked using the `XPU_IMAGE` macro:

```
1 struct DeviceLib {};  // Unique type to identify the device library
2 XPU_IMAGE(DeviceLib);  // Must be called exactly once per library
```

Basic Workflow

Using xpu typically involves the following steps:

1. Initialize the runtime:

```
xpu::initialize();  // Use default settings
// Or with custom settings:
xpu::settings settings;
settings.device = "cuda0";  // Select specific device
xpu::initialize(settings);
```

2. Create a command queue for execution:

```
xpu::queue queue;  // Uses default device
// Or for a specific device:
xpu::queue queue(xpu::device{"cuda0"});
```

3. Allocate memory:

```
// Allocate memory on host and device for transfer
xpu::buffer<float> input(1024, xpu::buf_io);
// Allocate memory only on device
xpu::buffer<float> internal(1024, xpu::buf_device);
```

4. Transfer memory and launch kernels:

```
// Copy 'input' buffer from host to the device
queue.copy(input, xpu::h2d);
// Launch kernel
queue.launch<SomeKernel>(xpu::nblocks(1024), input, internal);
queue.wait(); // Wait for kernel to finish
```

The details of memory management and kernel declaration are covered in Sections 4.3.2 and 4.3.4 respectively.

CMake Integration

xpu is added to CMake-based projects as a regular subproject. This can be done by calling `add_subdirectory` or using the `FetchContent` module:

```
include(FetchContent)
FetchContent_Declare(xpu
  GIT_REPOSITORY https://github.com/fweig/xpu
  GIT_TAG        master
)
FetchContent_MakeAvailable(xpu)
```

xpu provides CMake integration through the `xpu_attach` function. This function handles the compilation of device code for all enabled backends:

```
add_executable(my_app ${sources})
xpu_attach(my_app ${device_sources})
```

Backends are enabled by setting the corresponding CMake options:

```
set(XPU_ENABLE_CUDA ON)  # Enable CUDA backend
set(XPU_ENABLE_HIP ON)   # Enable HIP backend
set(XPU_ENABLE_SYCL ON)  # Enable SYCL backend
```

The CPU backend is always enabled and serves as a fallback for development and debugging.

Another notable CMake option is `XPU_DEBUG`, which compiles device code without optimizations and debug information enabled.

In addition to the xpu-library, a small utility application called `xpuinfo` is always compiled alongside xpu. This can be used to list all available devices found by xpu.

Environment Variables

Most options passed to `xpu::initialize` can also be overwritten via environment variables:

- `XPU_DEVICE`: Overwrite the selected default device.
- `XPU_VERBOSE`: Switch to toggle internal logging. Displays information about device operations like memory allocation, kernel launches and memory transfers. Useful for debugging.
- `XPU_PROFILE`: Toggle to enable xpu timers. See also Section 4.3.3.
- `XPU_EXCLUDE`: Comma seperated list of backends that should be disabled at runtime.

4.3.2 Memory Management

Memory management in heterogeneous computing requires careful consideration of different memory spaces and data movement between them. While some frameworks like SYCL attempt to abstract this complexity away through automatic memory management, this can lead to reduced control and potential performance issues. xpu takes a different approach, providing RAII-based memory management while maintaining explicit control over allocations and data movement.

In principle, much of the complexity of GPU memory management is solved with the usage of managed memory. In this case, both host and device can access memory via the same address. The device driver migrates memory pages through page faults as

needed. This eliminates the need for explicit copy operations and allows GPU memory to be handled like regular host memory. While this can simplify GPU development, it comes at significant performance costs. Compared to direct memory management operations can be slower by a factor of 10 to 100. Even worse serialization in the driver can create a further bottleneck for parallel workloads [11]. Given the memory intensive nature of most tasks in HEP processing, this makes managed memory as the primary mechanism for handling device memory unacceptable. xpu also supports managed memory but this type of allocation is not the default and must be explicitly chosen by the user.

The core of xpu's memory management is the buffer class, which stores only a pointer to device memory. All additional information, including reference counting, is managed by the xpu runtime outside the buffer object. This design ensures that buffers can be used like raw pointers in device code with zero overhead. Upon buffer creation, the type of memory allocation must be specified:

- `device`: Memory allocated on the current device, not accessible from host.
- `pinned`: Memory allocated on host that is also accessible from the device.
- `managed`: Memory accessible from both host and device, with automatic synchronization.
- `io`: Memory allocated on host and device. Additionally eliminates double allocations and copy operations when running on CPU.

To access buffer data, xpu provides two view classes:

- `view`: For device-side access, can be constructed from a buffer or pointer with size.
- `h_view`: For host-side access. Retrieves the corresponding host buffer from the runtime and throws an exception if the memory is not host-accessible.

Both view classes provide similar interfaces to `std::span` in C++20, but with different safety guarantees. The device-side view includes range checking through device-side assertions in debug mode. The host-side h_view always performs range checking for safety, with an unsafe_at method provided for performance-critical sections where bounds checking overhead needs to be eliminated.

This design provides several advantages:

- No hidden allocations or copies that could impact performance.
- Clear distinction between device and host access patterns.
- Support for zero-copy operations through io buffers.
- RAII-based cleanup of allocations.
- Zero overhead in device code through lightweight buffer objects.

- Type-safe access to memory with optional bounds checking.

This contrasts with SYCL's approach, which offers two different memory models. The buffer API requires the creation of accessors for every kernel that needs to access a buffer, leading to significant boilerplate code. While this enables automatic task parallelization, it comes at the cost of verbose code and limited control over allocations and copy operations. The USM (Unified Shared Memory) API introduced in SYCL 2020 offers direct allocation control but provides only C-style manual memory management without RAII support.

xpu's approach combines the safety of RAII with the control of manual memory management, making it suitable for applications where predictable performance and explicit memory control are essential.

4.3.3 Profiling API

Performance analysis of GPU applications requires tracking multiple types of operations: kernel execution, memory transfers, memory initialization, and overall wall time. xpu's profiling API provides an integrated way to collect these metrics with minimal code instrumentation and without the need to use external tools.

The core of the API is the timer concept. Timers can be started explicitly or created as scoped objects:

```
// Explicit timer management
xpu::push_timer("processing");
process_data();
auto timing = xpu::pop_timer();

// RAII-based alternative
{
  xpu::scoped_timer timer("processing", &timing);
  process_data();
}
```

Each timer automatically collects:

- Total wall time
- Time spent in each kernel, including per-invocation times
- Duration of host-to-device and device-to-host transfers
- Time spent in memset operations

Timers can be nested to provide hierarchical profiling information:

```
xpu::push_timer("full_pipeline");
{
  xpu::push_timer("preprocessing");
  prepare_data();
  auto prep_time = xpu::pop_timer();

  xpu::push_timer("computation");
  run_algorithm();
  auto comp_time = xpu::pop_timer();
}
auto total_time = xpu::pop_timer();
```

To enable performance analysis in terms of throughput, the API allows specifying the amount of data processed:

```
// Record bytes processed in current timer scope
xpu::t_add_bytes(input_size);

// Record bytes processed by specific kernel
xpu::k_add_bytes<MyKernel>(input_size);
```

This information is used to automatically calculate processing rates in GB/s, accessible through the `throughput()` methods of the `timings` and `kernel_timings` classes.

The profiling system is designed to have very little overhead when disabled. Only wall time measurements are collected unless profiling is explicitly enabled during initialization:

```
xpu::settings settings;
settings.profile = true;
xpu::initialize(settings);
```

When profiling is enabled, the system introduces implicit synchronization points after each kernel launch and memory operation to collect accurate timing information. This means that operations that would normally execute asynchronously are forced to complete before the next operation begins. While this provides precise timing data, it can significantly impact the overall performance of the application by preventing concurrent execution. Users should be aware of this overhead and disable profiling for production runs where maximum performance is required.

All timing data is accessible through the `timings` and `kernel_timings` class, allowing for programmatic analysis of performance data or integration with external monitoring systems.

```
#include <xpu/device.h>

struct VectorAdd : xpu::kernel<DeviceLib> {
  using block_size = xpu::block_size<256>;
  using context = xpu::kernel_context<>;
  XPU_D void operator()(
    context &ctx,
    xpu::buffer<const float> a,
    xpu::buffer<const float> b,
    xpu::buffer<float> c,
    size_t N);
};
```

Figure 4.2: Example of a kernel definition in xpu.

4.3.4 Device-side API

The following sections provide an overview of device-side functionanlity in xpu.

Kernel Definition

Kernels in xpu are implemented as callable objects that inherit from the `xpu::kernel` class. For this, every kernel must overload the function call operator marked with the `XPU_D` macro for device code compilation. The first argument of this operator is always a kernel context object, which provides access to the thread position, shared memory, and constant memory.

The kernel class can specify several optional properties through member type definitions:

- `block_size`: Sets the number of threads per block at compile time.
- `shared_memory`: Defines the shared memory layout for the kernel.
- `constants`: List of values in constant memory that the kernel needs to access.
- `context`: Can be used as a shorthand to define the kernel's context type, combining shared memory and constant memory requirements.
- `openmp`: Controls the OpenMP [30] execution settings when running on CPU.

Additional kernel arguments following the context parameter can be of any type that is copyable to the device. For frequently used types, xpu provides specialized buffer objects that manage device memory. When passing buffers to kernels, their size can often be inferred from other parameters or the grid size, reducing register usage in device code. Plain pointers are also supported as kernel arguments, though the user is then responsible for ensuring the memory is accessible on the device.

The kernel implementation must be marked with `XPU_EXPORT` to register it with the runtime system. This enables xpu to dynamically load the appropriate implementation for the active backend at runtime. For the CPU backend, the kernel is compiled as a regular C++ function using OpenMP for parallelization, with the thread indexing system adjusted to mimic GPU behavior.

An example for a basic kernel declaration is given in Figure 4.2. The kernel receives three buffers a, b, c and the buffer size N as arguments. Additionally kernels can specify the block size at compile time via the `block_size` member type. The first argument of each kernel is required to be a context object. The context contains the thread position, a pointer to the shared memory and access to constant memory if required.

Note: xpu also supports templated kernels. However, due to the separate compilation of host and device code, kernels and the corresponding host code are never in the same compilation unit. Thus every kernel template must be instantiated explicitly with a call to `XPU_EXPORT`. This can lead to a lot of boilerplate for heavily-templated code. A way to mitigate this issue in xpu is calling these kernels through host functions (see Section 4.3.6) instead.

Grid Position

Each kernel receives a context object that provides information about the thread's position in the execution grid. The position is defined by three coordinates in a hierarchical system: the position of the thread within its block (`thread_idx`), the dimensions of the block (`block_dim`), and the position of the block within the grid (`block_idx`). These coordinates can be accessed individually for each dimension (x, y, z) through methods like `thread_idx_x()` or `block_dim_y()`. Additionally, the total grid dimensions are available through `grid_dim_x/y/z()`. This system directly maps to CUDA/HIP's built-in `threadIdx`, `blockDim`, and `blockIdx` variables. On CPU, these functions reflect a grid layout with blocks of size 1.

Shared Memory

Shared memory in xpu is declared as a nested class named `shared_memory` within the kernel. This class is allocated into shared memory at kernel launch and can be accessed through the context object via the `smem()`-method. While this approach requires the shared memory size to be known at compile time, it enables straightforward usage of C++ unions in cases where device occupancy is limited by shared memory size. If no shared memory is required, the kernel can use `xpu::no_smem` as its shared memory type.

4.3.5 Constant Memory

Constant memory is a special type of memory in GPU programming that provides fast, read-only access to data that remains unchanged during kernel execution. While this feature is natively supported by CUDA and HIP, it was deprecated in SYCL in the 2020 specification [40]. However, constant memory remains valuable for many applications, particularly in high-energy physics where it can efficiently store calibration data or detector geometry information. To address this, xpu provides a unified abstraction for constant memory that works consistently across all supported backends.

Constants in xpu are declared as types that inherit from `xpu::constant`, specifying both the device library they belong to and their data type:

```
struct CalibrationData : xpu::constant<DeviceLib, float[1024]> {};
XPU_EXPORT(CalibrationData);
```

This declaration creates a type that represents a constant array of 1024 floating-point values. The first template parameter associates the constant with a specific device library, enabling separate compilation of device code, while the second parameter defines the type of data to be stored. Like kernels, constants must be registered with the runtime system using `XPU_EXPORT`.

To use constants in a kernel, they must be explicitly declared through the kernel's `constants` type:

```
struct Calibrate : xpu::kernel<DeviceLib> {
  using constants = xpu::cmem<CalibrationData>;
  using context = xpu::kernel_context<xpu::no_smem, constants>;

  XPU_D void operator()(context& ctx, xpu::buffer<float> data) {
    float calibration = ctx.cmem<CalibrationData>()[idx];
    data[idx] *= calibration;
  }
};
```

The `constants` type definition uses `xpu::cmem` to specify which constants the kernel requires access to. Multiple constants can be specified if needed:

`xpu::cmem<Constant1, Constant2, ...>.`

Within the kernel, constants are accessed through the context object, either by requesting a specific constant with `ctx.cmem<Constant>()` or accessing the entire constant memory object with `ctx.cmem()`.

On the host side, constant values are set using the `xpu::set` function:

```
float calibration_data[1024] = { /* ... */ };
xpu::set<CalibrationData>(calibration_data);
```

The implementation internally handles the mapping to appropriate backend-specific features. For CUDA and HIP, it uses native constant memory. For SYCL, where constant memory is no longer available, constants are instead written to global memory. This ensures that code written using xpu's constant memory abstraction remains portable across different GPU architectures while retaining optimal performance where native support exists.

Math Functions

xpu supports the overlap of math functions supported by CUDA, HIP, SYCL and the C++ standard library. Exceptions are functions where trivial fallbacks are possible. Some examples for this are the `cospi` and `sinpi` functions that are not available in the C++ standard library but are supported by the other backends available in xpu.

Atomic Operations

xpu provides atomic operations that work consistently across all supported backends. The standard set of atomic operations (compare-and-swap, addition, subtraction, bitwise AND, OR, and XOR) are available for 32-bit integers, with atomic compare-and-swap and addition also supporting single-precision floating point values.

For improved performance when threads are known to be in the same block, block-scoped variants of these operations are provided with the `_block` suffix. On GPUs, these are compiled to block-scoped intrinsics when supported by the architecture, falling back to regular atomic operations otherwise. On CPU, where blocks have a size of 1, these operations are compiled as regular non-atomic operations since synchronization is not needed in this case.

Block- and Warp-wide Functions

For efficient parallel processing, xpu provides a set of common parallel primitives that operate across warps and thread blocks. These operations include reductions, scans (prefix sums), and ballot operations. The implementation automatically selects optimal native functions for each backend while providing portable fallbacks for CPU execution.

4.3.6 Native Host Functions

By design, host code in xpu is seperated from device code and has to go through the xpu runtime to start kernels. However, in some cases it can be beneficial to compile host code with the GPU compiler and access device kernels directly. For this reason xpu

supports integrating native GPU library functions through its function mechanism. Similar to kernels, functions are declared by inheriting from `xpu::function` and must be registered with `XPU_EXPORT`. This allows platform-specific implementations to be compiled separately for each backend and selected at runtime.

This feature is particularly useful for two scenarios: First, when integrating heavily-templated kernels where the template instantiation should happen in the GPU compilation unit instead of the host code. Second, when using native libraries like CUB [50] or hipCUB [43] that provide optimized device-wide operations. The function abstraction provides type safety and seamless integration with xpu's buffer management while maintaining the performance characteristics of native implementations.

4.4 Mixing Compilers and Dynamic Loading

One of xpu's key design goals is supporting multiple GPU architectures while using each vendor's optimized compiler for their respective platform. This differs from approaches like Alpaka that support all platforms available to the current compiler, requiring device code to be templated to comply with the One Definition Rule (ODR) [51]. Instead, xpu compiles device code separately for each backend, allowing different compilers to be used for different platforms. While this approach provides optimal performance for each target, it creates the challenge of managing multiple compiled versions of the same functions and selecting the appropriate version at runtime.

The POSIX standard provides basic facilities for loading code at runtime through functions like `dlopen`, `dlsym` and `dlclose`. However, these functions were designed for C and present several challenges when working with C++ code. Due to name mangling in C++, there's no portable way to derive a symbol name from a function name. And reverting to C identifiers for kernel names would complicate the usage of namespaces and templated kernels.

Another approach would be to call `dlopen` multiple times for different backends to switch between implementations on the fly. However this has multiple drawbacks when using devices with different backends at once. The relocations can cause a lot of overhead when switching backends. Furthermore this is a global operation so it's not thread-safe either and would require additional synchronization.

To address these limitations, xpu implements a custom symbol table mechanism, illustrated in Figure 4.3 . Internally each device library maintains its own symbol table mapping types to function pointers. The symbol table is exported through a C interface accessible via `dlsym`. A custom type information system maps types to numeric identifiers (0...N) instead of using STL's `std::type_info` [1] which provides hash codes

[1] `std::type_info`'s hash and type name are implementation defined. This makes it unusable between compilers and for introspection. Libraries that rely on this functionanlity usually instead implement their own version relying on template and C-macro tricks. One notable `type_info` implementation can be found in the `entt` library [16].

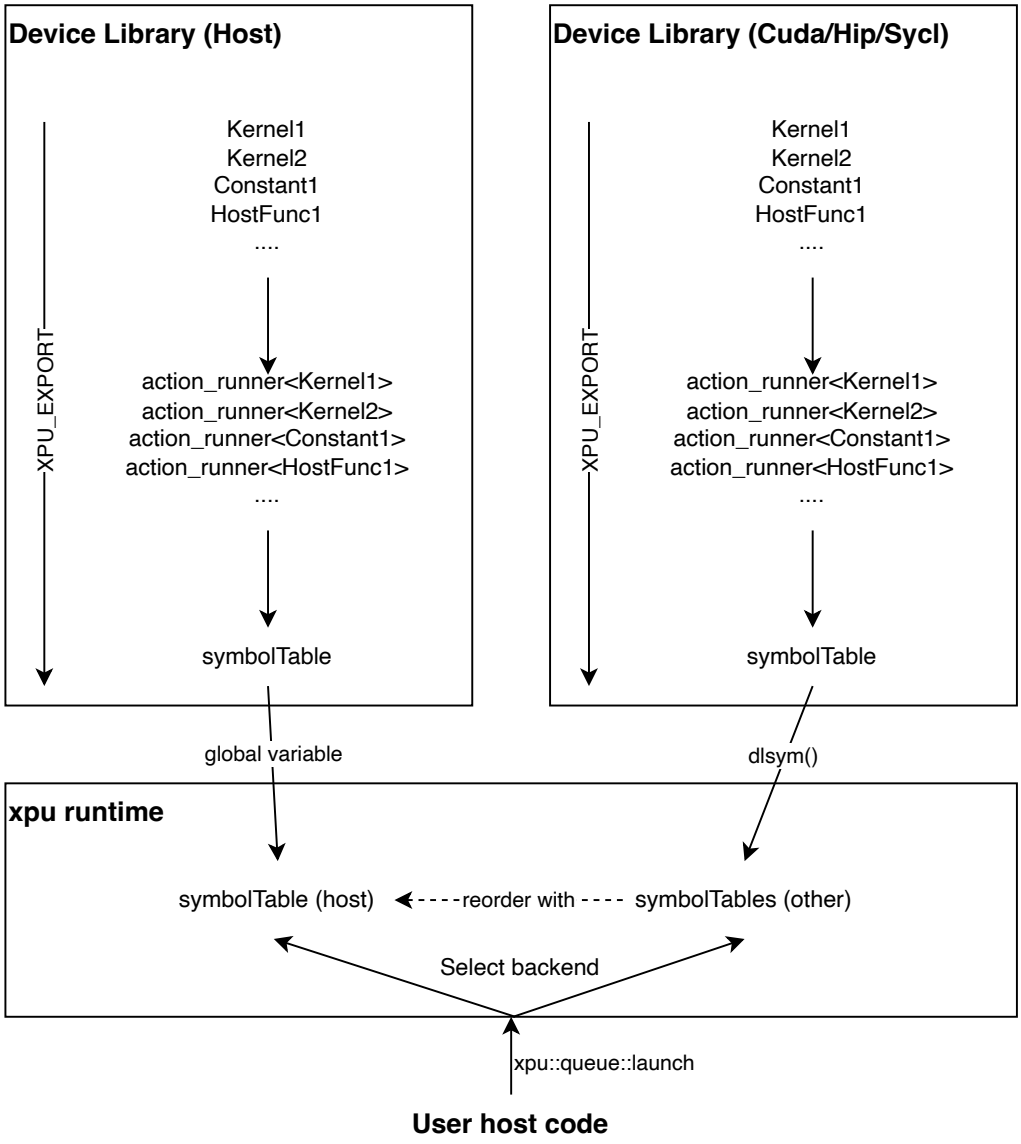

Figure 4.3: Concept how kernels, constants and host functions in xpu device libraries are propagated to the runtime and enable calling from user host code.

that aren't guaranteed to be consistent across compilers.

The implementation uses functors rather than plain functions, providing several advantages:

- Additional compile-time information can be attached (e.g., block sizes for kernels).
- Type safety is maintained across the dynamic loading boundary.
- Function addresses can be registered during static initialization.

Here's how xpu components are registered with the symbol table:

```
// Kernel definition
struct MyKernel : xpu::kernel<DeviceLib> {
  XPU_D void operator()(context& ctx, /*...*/) { /* Implementation */ }
};

// Host function definition
struct MyHostFunc : xpu::function<DeviceLib> {
  int operator()(/*...*/) { /* Implementation */ }
};

// Constant definition
struct MyConstant : xpu::constant<DeviceLib, float[1024]> {};

// Register with symbol table
XPU_EXPORT(MyKernel);    // Register kernel
XPU_EXPORT(MyHostFunc);  // Register host function
XPU_EXPORT(MyConstant);  // Register constant
```

The registration is handled by `register_action`, which provides specialized implementations for kernels, host functions, and constants:

```
template<typename A, xpu::driver_t D>
struct register_action {
  using image = typename A::image;
  using tag = typename A::tag;

  register_action() {
    // Register with the symbol table based on the action type
    if constexpr (std::is_same_v<tag, kernel_tag> ||
                  std::is_same_v<tag, function_tag>) {
      symbol_table::instance<image, D>().template add<A>(
          (void*)&action_runner<tag, A, decltype(&A::operator())>::call);
    } else if constexpr (std::is_same_v<tag, constant_tag>) {
      symbol_table::instance<image, D>().template add<A>(
          (void*)&action_runner<tag, A>::call);
    }
  }

  static register_action<A, D> instance;
};

template<typename A, xpu::driver_t D>
register_action<A, D> register_action<A, D>::instance{};
```

The XPU_EXPORT macro then simply expands to a template instantiation of `register_action`:

```
#define XPU_EXPORT(name) \
  template struct xpu::detail::register_action<name, \
    XPU_DETAIL_COMPILATION_TARGET>
```

4.4.1 Runtime Symbol Resolution

When a component is accessed through the xpu runtime, the following process occurs:

1. The runtime looks up the active backend's symbol table.
2. The type is used to find the corresponding function pointer.
3. For kernels and functions, arguments are passed to the implementation via variadic templates.
4. For constants, the value is written to the appropriate memory location.

This approach allows for efficient switching between implementations without reloading libraries. While also supporting multiple backends simultaneously and maintaining type safety across compilation boundaries. The symbol resolution overhead is

minimal as it occurs only once during initialization, with subsequent accesses reading single values from an array of function pointers.

4.5 Performance Overhead

In this section, the potential performance overhead with xpu and SYCL is investigated. The GPU port of the STS unpacker was chosen as the basis for the benchmark. The unpacker is first step in the processing chaing and decontextualizes the stream of raw detector data which is required for the subsequent cluster finding. The details of the unpacker are discussed in the master thesis of S. Heinemann [42]. The unpacker kernel was chosen for the following reasons:

- It is highly memory bound, making it sensitive to potential inefficient ordering of memory instructions by the compiler.
- It is also simple and small to allow enough equivalent implementations across multiple frameworks. This narrows potential performance factors down to the device compiler and native device functions.

The initial implementation of the unpacker was written in xpu. Additionally a native CUDA and SYCL port were added for comparison. The HIP implementation is generated from CUDA via the ROCm `hipify` script. SYCL was compiled with AdaptiveCpp 24.06 and Intel's `icpx` compiler shipped with OneApi 2024.2.1. Additionally ROCm version 6.0 and CUDA version 12.6.0 were used. Tests were conducted on a RTX 2080 Ti and Mi50 for Nvidia and AMD hardware respectively. 30 timeslices from the mCBM 2022 campaign were used as testdata. As the unpack kernel is very short, < 1 ms to a few milliseconds depending on the size of the timeslice, each timeslice was processed 1000 times consecutively. The benchmark shows the accumulated time of all kernel calls.

Performance numbers are compared in figure 4.4. The xpu version is essentially able to match the performance of both native implementations, with only a 0.2 % overhead in runtime compared to the respective CUDA and HIP variants. The generated assembly is almost identical. Figure 4.5 shows an example for differences in the machine code. For both the native CUDA implementation and xpu identical instructions are generated, only their ordering is different is this case.

The performance differences between SYCL implementations can be traced back to their distinct compiler architectures. Intel oneAPI and AdaptiveCpp both build upon LLVM / Clang but take fundamentally different approaches. Intel oneAPI uses a heavily customized Clang fork with proprietary optimizations, which explains its more consistent but slower performance across platforms - about 36% slower on CUDA with a similar performance gap on ROCm. AdaptiveCpp, compiled with Clang 18, uses a plugin architecture to generate backend code. While this approach promises better maintainability and easier updates to new Clang versions, the performance results

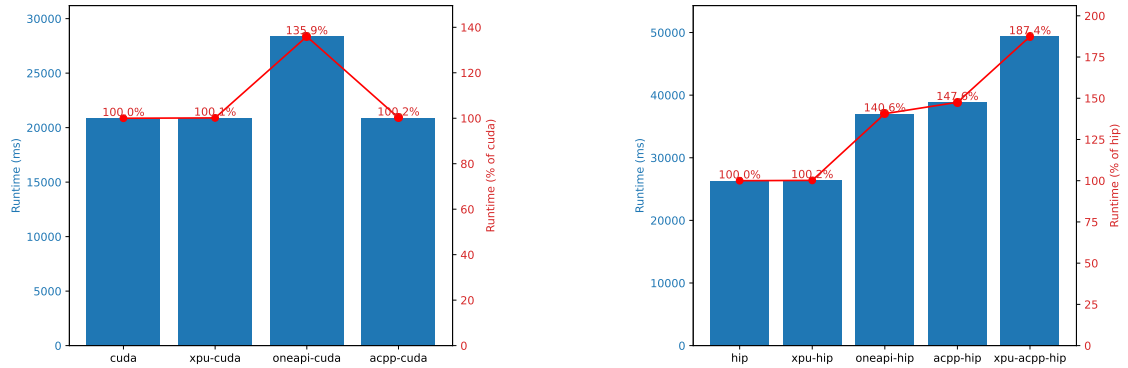

Figure 4.4: Performance comparison of different implementations of the STS unpacker. The left plot shows multiple implementations targeting Nvidia using the native CUDA implementation compiled with nvcc as baseline. Likewise implementations targeting AMD are shown on the right with the native HIP implementation compiled with amdclang as baseline.

Offset	Cuda compiled via nvcc	xpu compiled with nvcc
1a20	LDS.U R16, [0x8a4] ;	LDS.U R16, [R18.X4+0x8a8]
1b30	LDS.U R13, [R18.X4+0x8a8]	LDS.U R13, [0x8a4] ;
1b50	IADD3 R33, P2, P3, R16, R6	IMAD.SHL.U32 R23, R23, 0x2
1b60	IMAD.SHL.U32 R23, R23, 0x2	LOP3.LUT R28, R28, 0x1ff,
1b70	LOP3.LUT R28, R28, 0x1ff,	LOP3.LUT R28, R23, 0x200,
1b80	IADD3.X R31, RZ, R7, R31,	IMAD.MOV.U32 R25, RZ, RZ,
1b90	LOP3.LUT R28, R23, 0x200,	IADD3 R33, P2, P3, R13, R6
1ba0	IMAD.MOV.U32 R25, RZ, RZ,	IADD3 R13, P4, R16, -c[0x0
1bb0	IADD3 R13, P4, R13, -c[0x0	IADD3.X R31, RZ, R7, R31,

Figure 4.5: Example of the differences in generated assembly by nvcc when compiling CUDA code directly (left) versus compiling it via xpu (right). In this case, identical instructions are generated but in a different order.

suggest that the translation layer from SYCL to native GPU code introduces significant overhead, particularly for AMD hardware where the implementation runs 47% slower than native HIP.

Beyond pure performance considerations, both SYCL implementations exhibited issues when integrated with xpu. While both compilers successfully compile the test kernel when using SYCL directly, the additional abstraction layer in xpu revealed stability problems. The oneAPI compiler failed to generate correct code for the kernel when compiled through xpu, producing no results and ignoring debug output. AdaptiveCpp's ahead-of-time compilation worked for HIP but crashed at runtime when targeting CUDA via xpu. Its just-in-time compilation mode, which enables additional backends like OpenCL and Level Zero, lacks support for the block-wide scan operations required by the kernel, preventing its use in the benchmark entirely. These limitations also effectively prevented testing on Intel GPUs, where only the native Intel compiler is available.

It's important to note that this benchmark is limited in scope, focusing only on the STS unpacker kernel which represents a specific data access pattern. A recent study by Davis et a. [31] compared SYCL, CUDA, and HIP performance across a more broader set of benchmarks. Their results show SYCL matching or even slightly outperforming native implementations in many cases, suggesting SYCL's overhead may be less pronounced for more general computational patterns. However, how well either result can be generalised to the rest of the processing chain in CBM, in particular for track reconstruction, remains future work.

Chapter 5

GPU-Accelerated STS Reconstruction in CBM

5.1 Introduction

The Compressed Baryonic Matter (CBM) experiment, currently under construction at the Facility for Antiproton and Ion Research (FAIR) at GSI, aims to explore the properties of strongly interacting matter under extreme conditions of temperature and density via heavy-ion collisions. The Silicon Tracking System (STS) reconstruction in CBM faces significant computational challenges due to its free-streaming data acquisition approach. Unlike traditional triggered systems, CBM must process the complete raw data stream in real-time at interaction rates up to 10 MHz. As the primary tracking detector, the STS plays a crucial role in track reconstruction, making its efficient online processing critical for real-time event selection.

The existing CBM software framework, CBMRoot, was primarily developed with detector simulation and offline analysis in mind. Built on ROOT, it provides an environment for physics analysis but its deep integration with ROOT and focus on simulation and event-based processing make it less suitable for online reconstruction of free-streaming data. To handle the expected data rates in real-time, a new sub-project within CBMRoot was created, that avoids these constraints [36]. The design eliminates dependencies on ROOT in its core algorithms, focusing instead on optimization for continuous data processing. A key architectural decision was the clear separation between algorithms and framework layers, allowing performance-critical code to be developed and optimized independently.

As part of this software rewrite, several key components were developed and integrated. A new executable was created to run the online reconstruction chain, integrating unpacking from all available subsystems, reconstruction algorithms from STS, TRD and TOF as well as track reconstruction. For real-time monitoring during data taking, an integration with InfluxDB and Grafana was implemented. The continuous integration system was extended to support container building for online reconstruction, and a new YAML-based configuration system was developed to handle both geometry and readout descriptions.

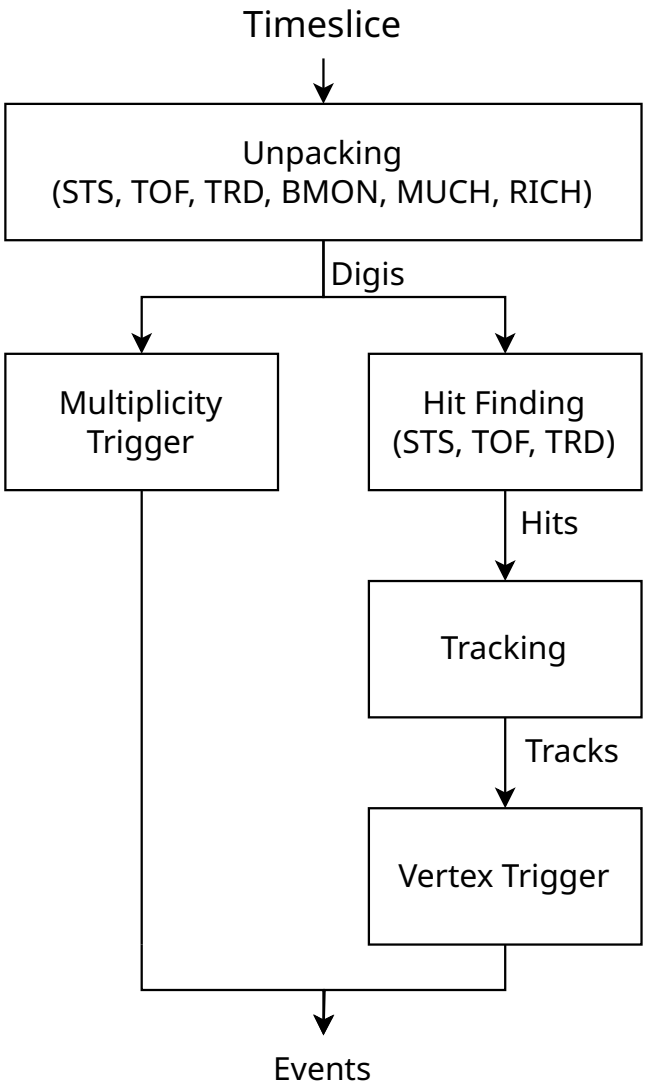

Figure 5.1: Overview of the CBM online processing chain. The workflow begins with unpacking raw data from all detector subsystems, producing digis that feed into either a simple multiplicity trigger path and a reconstruction path with hit finding, tracking, and vertex-based triggering. Both paths ultimately result in digi events.

As shown in Figure 5.1, the online processing workflow begins with unpacking raw data from multiple detector subsystems from timeslices. These timeslices represent fixed-duration intervals of continuous detector readout, typically spanning several milliseconds and containing multiple collision events. Unlike traditional event-based data acquisition, timeslices capture all detector signals within their time window regardless of trigger decisions. The resulting digis serve as input for one of two processing branches: a multiplicity-based trigger path and a more complex reconstruction path. The multiplicity trigger provides a computationally inexpensive first-level selection by simply counting the number of digis across detectors, allowing quick identification of events with sufficient activity. Alternatively, the reconstruction path performs hit finding for the tracking detectors (STS, TOF, TRD), followed by track reconstruction and vertex-based triggering. The vertex trigger offers more sophisticated event selection by identifying collision vertices and applying physics-motivated criteria such as vertex position and track multiplicity. Both paths result in digi events that can be stored for later physics analysis.

Within that context, this chapter presents the GPU-accelerated STS reconstruction developed for the online framework. New parallel algorithms for cluster and hit finding that exploit the parallelism available in GPUs are introduced. An abbreviated description of the GPU-accelerated hitfinder and testing of the online software through data challenges have also been published in the CBM Progress Report 2023 [73][29].

The chapter is organized as follows:

- Section 5.2 describes the existing offline STS reconstruction algorithm and its limitations.
- Section 5.3 presents the implementation of the GPU-accelerated algorithms for cluster and hit finding.
- Section 5.4 analyzes the performance characteristics of different GPU sorting strategies.
- Section 5.5 provides a performance evaluation of the complete STS reconstruction.
- Section 5.6 discusses the integration and validation through data challenges and deployment during the May 2024 mCBM beamtime.

5.2 The Offline STS Hitfinder

5.2.1 Overview

The Silicon Tracking System's reconstruction transforms raw detector measurements into four-dimensional spacetime points suitable for track reconstruction. As shown in Figure 5.2, the reconstruction pipeline processes input from two distinct sources: real

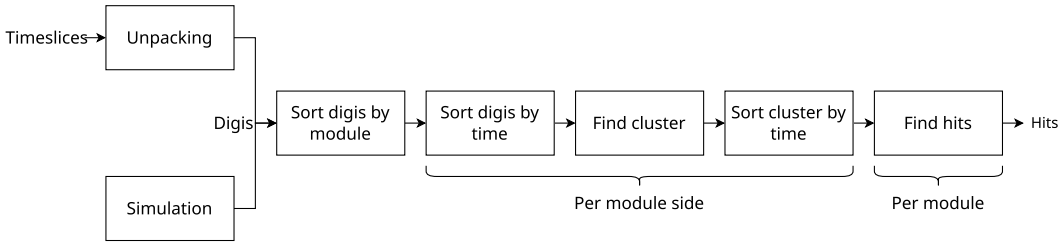

Figure 5.2: High level overview of the STS reconstruction steps.

detector data that must be unpacked from timeslices, or artificial data from detector simulation[1]. Both sources produce digis. These are time-stamped charge measurements carrying information about the deposited charge, the strip that collected it, and its timestamp.

The processing chain starts by distributing digis to their corresponding module side via a bucket sort. Afterwards digis are sorted by their timestamp as prerequisite for the following cluster finding.

Within each module side, the cluster finding algorithm identifies groups of adjacent strips that registered charge deposits within a configurable time window. These clusters capture the one-dimensional position along the strips through charge-weighted averaging, accompanied by timing information derived from the contributing digis. The resulting clusters are again sorted by time to prepare the hit finding stage.

The hit finding then combines time-matched clusters from both sensor sides of a module to reconstruct hit positions. When clusters from opposite sides are found within a matching time window, their intersection point determines the hit position in the sensor plane. The algorithm propagates measurement uncertainties from the initial charge measurements through clustering to the final hit position, providing error estimates for the reconstructed coordinates and timestamp. So the reconstructied hits are four-dimensional spacetime points that serve as input for subsequent track reconstruction.

At the highest level, this reconstruction process is modularized to handle data independently for each sensor module, enabling parallel execution via OpenMP [30] that was introduced by F. Boeck [14]. The core algorithms for cluster and hit finding were developed by V. Friese [37] and H. Malygina [48], respectively.

5.2.2 Cluster Finding

In its sequential form, the algorithm processes time-sorted digis by examining each measurement in order. When encountering a new digi, it checks whether an existing cluster is being formed on adjacent strips within a configurable time window. If such

[1] In this case, the simulation is not on raw data level.

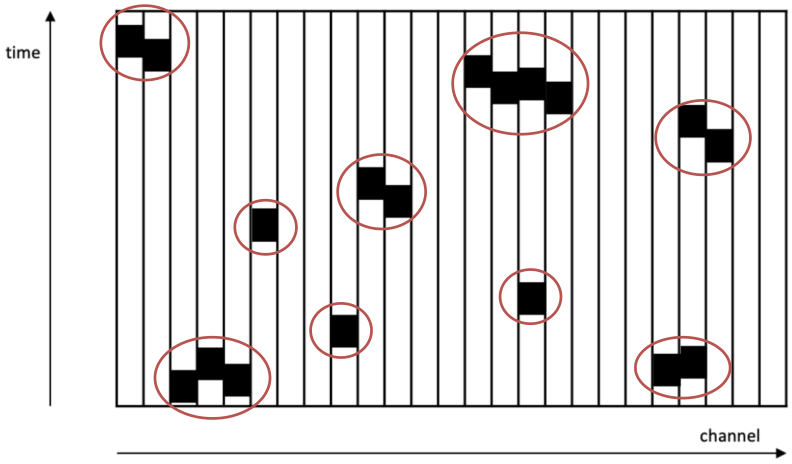

Figure 5.3: Visualization of ADC values recorded on one side of a STS module. The horizontal axis represents the detector channels (strips), while the vertical axis shows the time dimension. Each black square indicates a channel-time position where an ADC value above threshold was recorded. Red circles highlight groups of adjacent ADC values that are combined into clusters during reconstruction. (Graphic adapted from [37]. Circles added for emphasis.)

a cluster exists, the digi is added to it. Otherwise, a new cluster is started. This process continues until all digis have been processed, resulting in a collection of clusters characterized by their charge-weighted mean positions and timing information. An example of digis combined into clusters is shown in Figure 5.3.

The offline implementation splits this process into two steps. The first step identifies which digis belong together, storing indices of contributing digis for each cluster. This separation allows the second step to analyze the collected digis and compute the cluster properties. this two-step approach requires additional memory to store the digi indices and involves a second pass over the data.

While this sequential approach is effective, it presents several challenges for parallel implementation, particularly on GPUs. The algorithm fundamentally relies on examining time-ordered data sequentially, with each decision potentially affecting subsequent cluster formation. This dependency makes partitioning the work effectively across multiple threads difficult. The variable-length nature of clusters further complicates work distribution, as the number of digis that will form a cluster is not known in advance.

5.2.3 Hit Finding

The hit finding algorithm transforms two-dimensional cluster measurements (time and channel) from both sensor sides into three-dimensional hit positions (time, x and y position on the sensor).[2] At its core, the algorithm processes two sorted lists of clusters from the front and back sides of each sensor. For each potential cluster pair, the algorithm first checks whether their time measurements are compatible within a configurable window. For time-matched clusters, the algorithm calculates their geometric intersection points along the overlapping strips. The temporal matching significantly reduces the number of cluster combinations that need to be evaluated geometrically, improving computational efficiency.

The intersection calculation is performed in the sensor's local coordinate system. Each strip direction defines a line in this coordinate system, and the intersection of these lines determines the hit position. However, due to the periodicity of the strip pattern, multiple intersections may occur for a single cluster pair. These ambiguities, known as ghost hits, are an inherent feature of the strip geometry and must be handled by the subsequent track reconstruction stage.

For each reconstructed hit, the algorithm must also propagate the position uncertainties from the input clusters to the final hit coordinates. This error propagation considers both the uncertainties in the cluster positions and the geometric effects of the stereo angle. The resulting hit objects contain the reconstructed position in global coordinates, timing information derived from the contributing clusters, and the associated measurement uncertainties.

5.3 The Online STS Hitfinder

5.3.1 General Performance Considerations

The online STS hitfinder employs several fundamental optimizations focused on efficient data structures and memory management. These optimizations are particularly critical given that the reconstruction must process between 10^8 and 10^9 hits per timeslice in real-time during data taking. [3]

A primary optimization involves the complete redesign of the data structures used for reconstruction. The offline reconstruction uses a hierarchical class structure deeply integrated with ROOT [15]. Hit classes follow the inheritance chain

[2] Technically, both cluster and hit position have an additional dimension along the z-axis that represents the distance from the target. However, this value is given implicitly by the sensor position in world coordinates.

[3] Rough estimate assuming a timeslice size of at least 10 GB with the STS making up at least 25 % of the data volume. See [2, p. 32–34] and [2, p. 45] for estimates of detector data rates and targeted timeslice sizes.

`TObject` [69] → `CbmHit` [18] → `CbmPixelHit` [19] → `CbmStsHit` [20],

,while cluster classes use

`TObject` → `CbmCluster` [21] → `CbmStsCluster` [22].

This inheritance from `TObject` is necessary as the objects are stored in ROOT's `TClonesArray` containers [68] throughout the CBMRoot framework. While these structures could theoretically be optimized in the offline code as well, applying the changes retroactively is a very involved undertaking. In particular a large number of analysis tasks depend on the current class hierarchy and member variables of STS hits and clusters or use the base classes for polymorphy to implement common operations across detector classes. In turn this means any change to these classes would entail adapting or restructuring large parts of the code base.

The online hitfinder instead implements lightweight C++ structures that eliminate these dependencies. By removing the ROOT integration and flattening the inheritance hierarchy, the redesign significantly reduces the memory footprint of the basic data types used throughout the reconstruction chain. For instance, the size of a cluster object is reduced from 112 byte in the offline reconstruction to just 24 byte in the online version. Similarly, hit objects are reduced from 136 byte to 48 byte. [4]

These reductions are achieved through several optimizations. Where full double precision is not required for reconstruction, single-precision floating point values are used instead. Timestamps are stored as 32-bit integers rather than 64-bit floating point numbers. The storage of Monte Carlo information, which requires an additional 8 byte per object for pointer storage and is crucial for simulation studies but irrelevant for online reconstruction, is separated from the main data structures.

The encoding of sensor information is also optimized. While the offline reconstruction uses 32-bit integers to encode sensor addresses, in the STS digi, the sensor address can be encoded using just 17 bit [23]. Combined with packing the 5-bit ADC value and 10-bit channel, this allows the reduction of the digi size to 8 byte from previously 16 byte. By keeping the sensor address inside the digi, it was possible to apply this change to the offline digi, which in turn is now used in both offline and online code.

For hits and clusters, the online code takes a different approach to handling the sensor address by using specialized container classes

`PartitionedVector` [53] and `PartitionedSpan` [52].

[4] These sizes can likely be even further reduced. Most fields of the STS clusters and hits don't need full 32 bit floating point precision. E.g. for position and time error estimates 16 bit or even 8 bit fixed-point numbers are probably sufficient.

These containers, which conceptually follow the design of `vector` and `span` from the C++ standard library, partition data by hardware address, storing the address only once per partition rather than in each object. The container classes are general enough to be used across multiple detector systems in the online reconstruction. This currently includes both the Time-of-Flight (TOF) [32] and Transition Radiation Detector (TRD) [1], the other two detectors whose hit reconstruction was ported to the online code.

Another significant optimization involves the handling of digi indices during cluster creation. The offline reconstruction allocates additional memory to store indices of contributing digis for each cluster, requiring both the memory for a `std::vector` object (24 byte), and an additional heap allocation. The online implementation eliminates this overhead by directly computing cluster properties during construction.

The reconstruction chain also exploits opportunities for increased parallelization. The offline version typically processes data at the module level, the online implementation can instead parallelize across module sides where appropriate, effectively doubling the available parallelism for these operations.

5.3.2 Efficient Sorting on GPU

The STS hitfinder performs sorting at multiple stages of its pipeline. First, incoming digis must be ordered by channel and time to enable efficient cluster finding. Later, clusters must be sorted by time to facilitate hit reconstruction.

The chosen GPU implementation divides the sorting task along natural partition boundaries - each GPU thread block processes data from a single module side independently. Within each block, a two-phase sorting approach is employed. First, the data is divided into fixed-size chunks that are sorted using a block-level radix sort. These sorted sequences are then merged in parallel using the Merge Path algorithm [38]. This parallel merge occurs entirely within the thread block, using shared memory to optimize memory access patterns. The parallel merge step was implemented in xpu (see Chapter 4) by K. Hunold as part of his master thesis [44].

The block-level radix sort phase processes data in fixed-size chunks, with each thread responsible for handling multiple elements. The implementation uses CUB's BlockRadixSort [27] which utilizes shared memory to minimize global memory accesses during the sorting process. Both the block size and the number of elements processed per thread are tunable parameters that affect the balance between device occupancy and shared memory usage. For the RTX 2080 Ti, optimal performance was achieved with 512 threads per block and 11 elements per thread, allowing each thread block to sort 5632 elements simultaneously. These parameters were chosen to maximize occupancy while staying within the device's shared memory constraints.

The parallel merge phase combines these sorted sequences using the Merge Path algorithm. For each merge operation, the algorithm first determines the appropriate partition points that split the sequences into roughly equal portions for each thread.

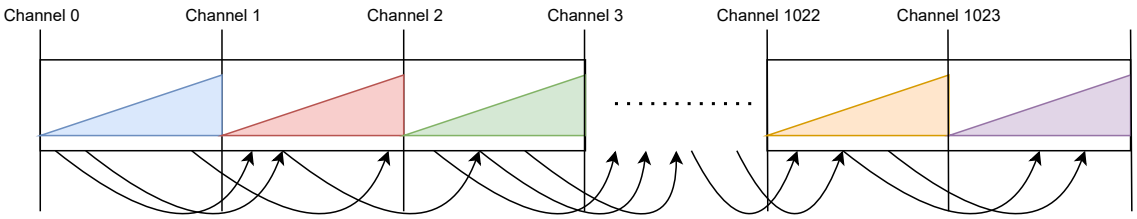

Figure 5.4: Illustration of the linked-list structure used by the parallel cluster finder, showing connections between digis in adjacent channels. Colored regions represent charge deposits in each channel with increasing time stamps, while arrows indicate connections established between digis that form clusters.

These partition points are computed in parallel using binary search, with each thread responsible for finding its own starting position in the merged sequence. The actual merging is then performed with each thread processing its assigned portion of the sequences, using shared memory as a temporary buffer to optimize memory access patterns.

The performance of this approach depends heavily on having sufficient partitions to fully utilize the GPU. Each partition (corresponding to a module side) is processed by a single thread block, requiring at least as many partitions as compute units on the device to achieve full occupancy. This is no issue for the full STS detector with its 1792 module sides but becomes a limitation for the mCBM test setup, where the mSTS has only 24 module sides and doesn't fully utilize modern GPUs. Section 5.4 discusses the performance implications of this and compares it to CUB's DeviceSegmentedSort [28], which would be an alternative sorting primitive that could be used in this case.

Concat Sort

In his master thesis [54], S. Sedighi developed a linear time sorting algorithm called "Concat Sort" for the STS data. This algorithm exploits the layout of microslices, where data from a single channel is always increasing in time but channel data may be interleaved. Thus only a reordering of data across channels is necessary. However, recent changes in the STS readout hardware broke these assumptions [75], whereby the algorithm isn't applicable anymore to newer STS data.

5.3.3 Parallel Cluster Finding

The parallel cluster finding algorithm was originally developed by K. Hunold for his master thesis [44]. His approach implements a two-phase strategy to enable efficient parallel processing on GPUs. Rather than attempting to identify complete clusters in a

single step, which would require complex synchronization between threads, the algorithm first establishes connections between digis that belong to the same cluster, then creates the actual cluster objects in a second phase.

At the core of this approach is the `DigiConnector` structure, which implements a simple linked list to track cluster membership. Each connector uses a single 32-bit value to store both the index to the next digi in the cluster and a flag indicating whether the digi has a predecessor. The highest bit is reserved for the predecessor flag, while the remaining 31 bit store the index of the next digi or zero to indicate the end of a cluster. This representation allows atomic updates to the entire structure using compare-and-swap operations, ensuring thread safety when multiple GPU threads attempt to modify connections simultaneously.

Hunold's implementation parallelized the workload by assigning one GPU thread block to each sensor side. Within each block, individual threads process digis in parallel. While this approach proved effective for the full STS detector with its 1792 module sides, it ran into limitations when processing data from the mCBM setup with only 24 module sides.

To address this issue, the algorithm was adapted to maximize GPU utilization on smaller detector configurations. The key modification was restructuring the implementation to enable parallelization across all digis regardless of their module assignment. Instead of organizing work at the module level, the cluster finding was decomposed into three kernels that each operates with one thread per digi:

1. A kernel that computes offsets for each channel, enabling efficient data access in subsequent steps.
2. The connection finding kernel that establishes links between digis.
3. The cluster creation kernel that traverses the connections to build clusters.

In the connection phase, each GPU thread processes a single digi, examining neighboring channels for potential cluster members. Two digis are considered part of the same cluster if they are in adjacent channels and their time difference falls within a configurable window. Figure 5.4 illustrates this process, showing how digis in adjacent channels are connected to form clusters. The colored regions represent charge deposits in each channel with increasing Timestamps, while the arrows indicate the connections established between digis. To efficiently find matching digis, the algorithm uses binary search to locate the first digi within the time window in the neighboring channel, taking advantage of the fact that digis are already sorted by channel and time. When a match is found, the thread atomically updates the `DigiConnector` to establish a forward edge to the next digi and sets the predecessor flag on the target digi.

The cluster creation phase follows a similar thread-per-digi approach. Each GPU thread that processes a digi without a predecessor (indicating the start of a cluster) is responsible for creating the corresponding cluster. The thread follows the chain of connections

established in the first phase, computing cluster properties like total charge, position, and timing information directly during traversal. This approach eliminates the need to store intermediate arrays of digi indices, reducing memory usage compared to the offline implementation to additional 4 byte per digi.

5.3.4 Parallel Hit Finding

The primary change to the hit finding algorithm was to increase the parallel work compared to the offline implementation. While the offline version processes data at the module level with one CPU thread per module, the GPU implementation instead assigns one thread per front-side cluster, enabling full parallelization across all potential hits.

For each front-side cluster, the assigned GPU thread searches for matching clusters on the back side of the sensor. To efficiently find potential matches, the algorithm exploits the fact that clusters are already sorted by time. The thread performs a binary search to locate the first back-side cluster within the configurable time window relative to its front-side cluster. This optimization significantly reduces the number of cluster pairs that need to be evaluated for geometric intersection, as clusters outside the time window can be quickly discarded.

The algorithm implements early exit conditions based on time differences between clusters. As both cluster arrays are sorted by time, the thread can stop searching once it encounters a back-side cluster whose time difference exceeds the maximum window. This approach, combined with the binary search for the initial matching cluster, means that each thread typically only needs to evaluate a small subset of back-side clusters.

Memory management for hit output employs a bucket-based approach, with preallocated space for each module sized according to the number of input clusters. Each thread uses atomic operations to obtain an index in the appropriate module's bucket when writing a new hit.

5.4 GPU Sorting Performance

5.4.1 Setup

The performance of the custom merge sort implementation was evaluated against CUB's `DeviceSegmentedSort` [28]. Two variants of the CUB implementation were tested: sorting key-value pairs and sorting keys only. For the pair sort, 8-byte keys are constructed containing the channel (2 byte) and timestamp (4 byte) with 2 byte of padding, while using the complete digis as values. The key-only variant requires modifying the digi format to store only channel, timestamp, and ADC value (8 byte total), dropping the 17-bit sensor address that is included in the current definition.

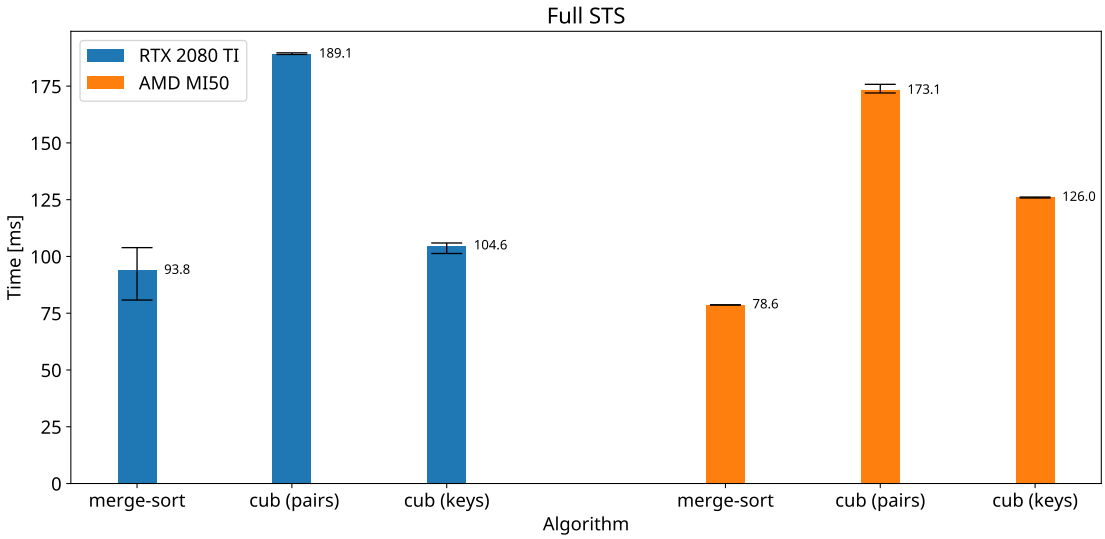

Figure 5.5: Synthetic benchmark to comparing GPU sorting algorithms on data from the full STS setup.

The performance measurements were conducted on two GPU architectures: an NVIDIA RTX 2080 Ti using CUDA 12.3 and an AMD MI50 using ROCm 6.0. For the AMD platform, the hipCUB port [43] of CUB's algorithms was utilized. The test dataset consists of approximately 1.5 GB of synthetic STS digis, chosen to occupy the majority of the RTX 2080 Ti's available memory with sorting buffers.

Two configurations representing different scenarios were examined: the full STS detector with 1792 segments (corresponding to module sides) and the mSTS test setup with only 24 segments. In both cases the dataset was uniformly distributed among these segments. This allows the evaluation of how the sorting approaches scale with different levels of parallelism. The performance measurements focus solely on kernel execution time. Each test configuration was run for 11 iterations, with the first iteration discarded as a warmup run. The reported results represent the median, maximum and minimum execution time of the remaining 10 iterations.

While the key-value pair sorting can serve as a direct replacement for the custom implementation, the key-only variant would require a bigger restructuring of the online code due to the removal of the sensor address from the digi format. The memory implications of these different approaches are discussed in detail in Section 5.4.4.

5.4.2 Full STS Sorting Benchmark

For the full STS detector configuration, Figure 5.5, the custom merge sort completes in 93.8 ms on the RTX 2080 Ti and 78.6 ms on the MI50, showing the fastest performance.

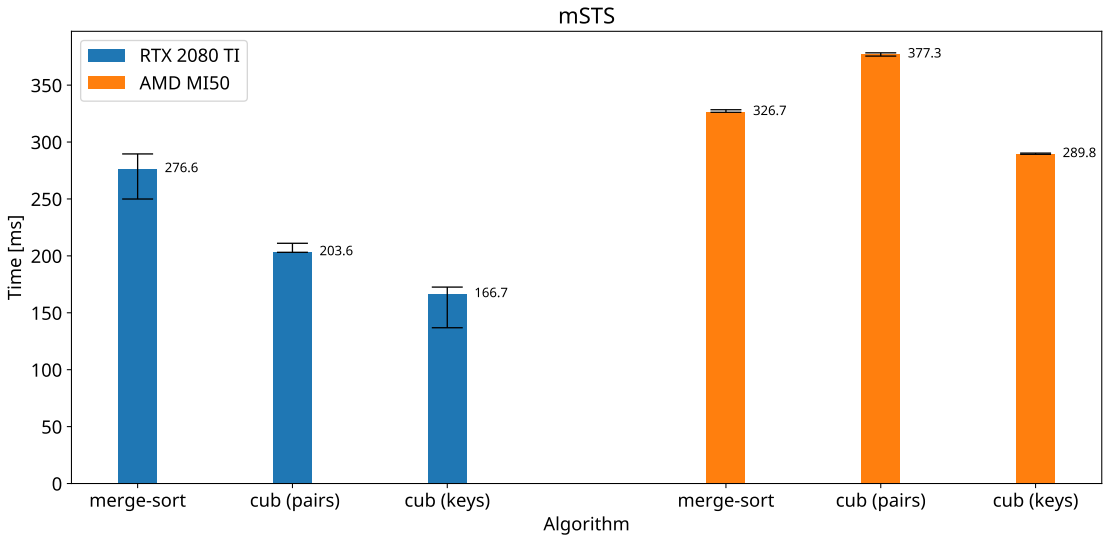

Figure 5.6: Synthetic benchmark to compare sorting algorithm on data from the mSTS setup.

In contrast, CUB's key-value pair sorting proves to be the least efficient approach, requiring approximately twice the execution time at 189.1 ms on the RTX 2080 Ti and 173.1 ms on the MI50. The key-only variant of CUB offers improved performance over pair sorting but still remains 10 % to 15 % slower than the custom implementation, taking 104.6 ms and 126.0 ms on the respective devices.

An interesting observation is comparing the performance across hardware platforms. While the custom merge sort achieves better performance on the MI50 compared to the RTX 2080 Ti, CUB's key-only sort shows the opposite behavior, running approximately 20 % slower on the AMD hardware. This suggests higher optimization of the CUDA implementation compared to the HIP port.

5.4.3 mSTS Sorting Benchmark

The performance characteristics change dramatically when processing data from the mSTS setup. As illustrated in Figure 5.6, in this scenario, the performance pattern inverts completely. CUB's key-only sorting emerges as the fastest option, completing in 166.7 ms on the RTX 2080 Ti, while the custom merge sort becomes the slowest at 276.6 ms. This performance degradation is even more pronounced on the MI50, where the merge sort requires 326.7 ms.

The custom merge sort's performance degrades by a factor of 3–4 on both devices compared to the full STS configuration. This is not unexpected, as the 24 module sides of the mSTS occupy only about one-third of the available compute units on either device (68

SMs on the RTX 2080 Ti and 60 CUs on the MI50). In contrast, the CUB variants maintain better performance by additionally parallelizing work within each partition.

5.4.4 Memory Usage

The memory requirements differ significantly between implementations. For n digis, the custom merge sort operates with $2n$ memory usage, requiring only an input and output buffer. Conversely, CUB's key-value pair sorting demands $6n$ memory, allocating space for two input buffers, two output buffers, and two temporary storage buffers (for keys and values respectively). The key-only variant of CUB reduces this to $3n$, needing only one buffer each for input, output, and temporary storage.

CUB's implementation enforces strict separation between input, temporary, and output buffers, prohibiting any overlap between these memory regions and preserving the input buffer's contents. On the other hand, the custom merge sort implementation reduces memory usage by modifying the input buffer during sorting, as the original order of the input data doesn't have to be preserved in this case.

5.5 Hitfinder Performance

5.5.1 Setup

To evaluate the performance improvements of the GPU-accelerated hitfinder implementation, a simulated dataset representing conditions of the full STS detector was used. The dataset consists of 1000 central Au+Au collisions, resulting in approximately 17.2×10^6 STS digis that produce 5.9×10^6 clusters and 4.8×10^6 reconstructed hits. The total input size of the dataset is approximately 130 MB.

Performance measurements were conducted on three different computing platforms. For GPU execution, an AMD MI50 and an NVIDIA RTX 2080 Ti were used. CPU scaling was evaluated on an Intel Xeon Gold 6130 running at 2.10 GHz. The CPU system features a dual-socket configuration with 16 cores per socket and two threads per core, providing a total of 64 hardware threads.

All measurements were performed on Ubuntu 22.04 using the GCC 11 compiler for host compilation. For GPU compilation and execution, CUDA 12.6 was used on the NVIDIA platform while ROCm 6.0 was employed for the AMD hardware.

Each test configuration was run for 11 iterations, with the first iteration discarded as a warmup run. The reported results represent the median execution time of the remaining 10 iterations or the breakdown of processing steps from that median run. Runtime measurements include both the complete reconstruction chain and timings

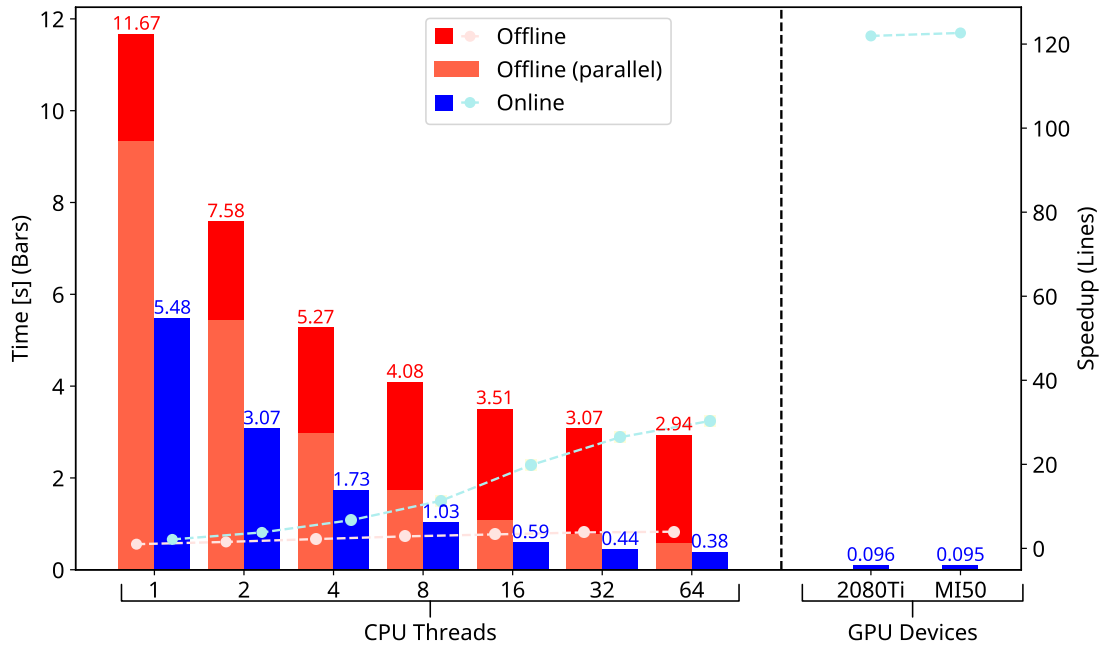

Figure 5.7: Performance comparison of the offline and online STS hitfinder.

of individual components to analyze the performance characteristics of different processing stages in the following sections. To allow consistent comparison, all speedup figures are calculated relative to the single-threaded execution time of the offline implementation.

5.5.2 Total Runtime

The overall performance characteristics of the hitfinder implementations are shown in Figure 5.7. For the offline implementation, the figure distinguishes between the parallel reconstruction phase (light red) and sequential overhead phases (dark red). These sequential phases comprise the initial distribution of digis to modules and the final collection of clusters and hits into contiguous output arrays.

When examining single-threaded performance, the offline implementation requires 11.67 s total runtime, with 9.4 s spent in the parallel section and 2.3 s in sequential overhead. The online implementation achieves better baseline performance at 5.48 s, representing a 2.1x speedup before any parallelization is applied.

Both implementations demonstrate effective scaling as the thread count increases. The parallel section of the offline version continues to scale up to 64 threads, showing that the reconstruction algorithm itself parallelizes well. However, the overall performance becomes increasingly dominated by the sequential overhead. This creates a

Processing step	Execution time				
	Off. CPU (1t)	Off. CPU (64t)	On. CPU (1t)	On. CPU (64t)	GPU
Sorting	2.4 s	0.16 s	1.22 s	0.069 s	0.006 s
Clustering	5.3 s	0.28 s	2.48 s	0.074 s	0.011 s
Hit Finding	1.63 s	0.1 s	1.19 s	0.065 s	0.009 s
Digi Pre-Sorting	1.34 s	1.56 s	0.287 s	0.021 s	0.017 s
Hit Collection	0.985 s	0.807 s	0.150 s	0.015 s	-
DMA Transfer	-	-	-	-	0.038 s

Table 5.1: Performance comparison of individual processing steps in the STS reconstruction chain. GPU measurements use the AMD MI50 times and show the accumulated kernel times for sorting, clustering and hit finding substeps.

performance bottleneck that limits the maximum achievable speedup for the offline implementation.

The online implementation addresses this limitation by parallelizing both the reconstruction and the data movement phases. This approach maintains scaling across the full range of tested thread counts, reaching 0.38 s at 64 threads for a 14.4x speedup from its sequential baseline. It should be noted that the relatively modest improvement when moving from 32 to 64 threads in both implementations is attributable to the use of simultaneous multithreading (hyperthreading), where the additional 32 threads run on the same physical cores as the first 32 threads, sharing execution resources.

GPU execution provides further substantial speedup. Both tested devices, the NVIDIA RTX 2080 Ti and AMD MI50, achieve similar performance of around 0.095 s. This represents approximately a 122x speedup compared to sequential offline execution and is 4.0x faster than the best CPU performance of the online version. The consistent performance across these different GPU architectures demonstrates the algorithm's ability to effectively utilize diverse hardware platforms.

5.5.3 Performance of Individual Processing Steps

Table 5.1 presents a breakdown of the execution time for individual processing stages in the STS reconstruction chain across different hardware configurations. This granular view reveals where performance gains originate and identifies remaining bottlenecks.

Examining the three core reconstruction steps (sorting, clustering, and hit finding), significant speedups are evident even on CPU-only execution. Detailed performance

scaling plots for these steps are provided in Appendix C.1. The online implementation achieves approximately a 2x speedup for sorting (2.4 s to 1.22 s), 2.1x for clustering (5.3 s to 2.48 s), and 1.4x for hit finding (1.63 s to 1.19 s) in single-threaded execution compared to the offline version. These improvements stem from the optimized data structures and algorithmic changes described in Section 5.3.1.

When comparing the best CPU performance (64 threads, online implementation) with GPU execution, additional substantial speedups are observed: approximately 11.5x for sorting, 6.7x for clustering, and 7.2x for hit finding. The clustering step benefits most from GPU acceleration in absolute terms, with execution time reduced from 2.48 s to just 0.011 s when comparing single-threaded CPU execution to GPU—a 225x improvement.

The digi pre-sorting and hit collection steps show different characteristics from the main reconstruction steps. In the offline implementation, these operations are not parallelized, explaining the minimal changes in runtime when increasing thread count. Although it remains unclear why these steps would be affected by the thread count at all and show a relatively large fluctuation in runtime.

The online implementation significantly improves the digi pre-sorting performance through two main optimizations. First, a hash map was replaced with a direct lookup table that uses the 17-bit address from digis to obtain the sensor index. Second, the online version implements proper parallelization for this step. On GPU, the digi pre-sorting benefits slightly from the fact that the pinned memory for digis is pre-faulted before processing begins, whereas on CPU the memory is allocated per timeslice, potentially causing page faults during execution that impact performance.

Hit collection in the offline version incurs significant overhead that scales poorly with thread count. The online CPU implementation reduces this time by a factor of 65x (from 0.985 s to 0.015 s) when comparing single-threaded to 64-threaded execution. On GPU, hit collection is effectively merged with the DMA transfer operation, eliminating it as a separate step.

The DMA transfer time (0.038 s) now constitutes more than one-third of the total GPU execution time, making it the dominant factor in overall performance. Combined with digi pre-sorting (0.017 s), these data movement operations account for approximately half of the total GPU runtime.

This bottleneck suggests several future optimization opportunities. The DMA transfer could potentially be overlapped with processing operations to hide latency. A more significant improvement would involve moving the unpacking step to the GPU, eliminating the need to transfer digis from host to device. This would be particularly beneficial since digis increase the data volume by approximately a factor of two compared to the raw timeslice data.

An architectural limitation in the current design relates to the dual use of digis. They serve as input for both hit reconstruction and the multiplicity trigger. While hit reconstruction requires digis to be sorted by sensor, the multiplicity trigger expects them to

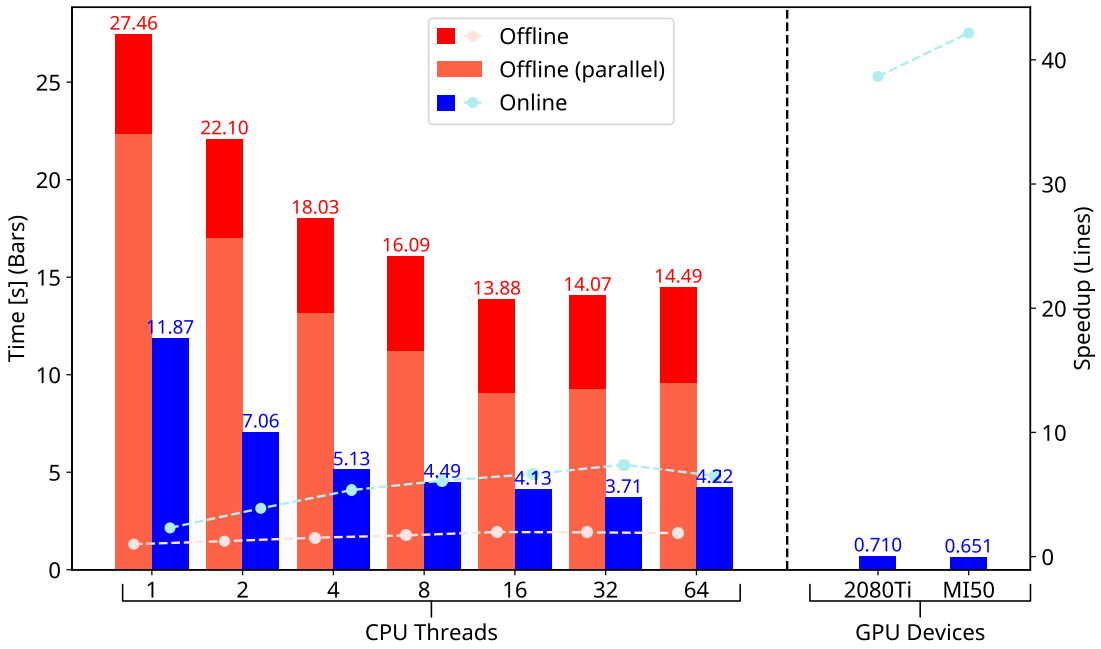

Figure 5.8: Performance comparison of the offline and online STS reconstruction on mCBM data.

be time-sorted. Resolving this conflict would require either maintaining two copies of the data in different sort orders or adapting the multiplicity trigger to support a different input format.

5.5.4 Performance on mCBM Data

While the GPU-accelerated hitfinder was primarily optimized for the full STS detector configuration, its performance was also evaluated on real data from the mCBM experiment. The test dataset consists of 20 timeslices from Run 2966 of the May 2024 mCBM beamtime, containing approximately 44.5×10^6 STS digis that produce 34.7×10^6 clusters and 1.4×10^6 reconstructed hits[5]. The total input size of the dataset is approximately 340 MB.

The mCBM setup represents a drastically scaled-down version of the full detector, with only 24 module sides compared to the 1792 in the full STS. This reduction creates a

[5] In comparison to the simulated dataset described in Section 5.5.1, the mCBM data contains more than twice the number of digis but produces approximately four times fewer reconstructed hits. This indicates that the real detector data likely contains significantly higher noise levels than accounted for in simulation.

challenging scenario for parallelization that was not specifically targeted during algorithm development. Figure 5.8 shows the accumulated processing time over all timeslices for both implementations.

In single-threaded execution, the offline implementation processes the dataset in 27.46 s, while the optimized online version completes in 11.87 s. This 2.3x speedup derives purely from the algorithmic improvements and data structure optimizations, matching the improvement ratio observed with full STS data.

The scaling characteristics with increased thread count reveal limitations specific to the mCBM configuration. The offline implementation's performance plateaus around 16 threads at 13.88 s, constrained by the limited parallelism available with only 12 modules. Although the online implementation maintains somewhat better scaling up to 32 threads, reaching 3.71 s, both versions demonstrate substantially worse scaling than observed with full STS data due to the inherently limited parallelism in the smaller detector configuration.

GPU acceleration still provides significant benefits, with processing times of 0.710 s on the RTX 2080 Ti and 0.651 s on the MI50. However, the speedup of approximately 40x over single-threaded CPU execution falls considerably short of the 128x improvement achieved with full STS data. This reduced relative gain stems primarily from the sorting stage limitations, where the 24 module sides occupy only about one-third of the GPU's compute units. The clustering and hit finding stages maintain better relative performance as they can parallelize work at finer granularity regardless of module count.

These results highlight how the optimizations, while not specifically targeted at small detector configurations like mCBM, still provide substantial performance improvements even in suboptimal parallelization scenarios. A more detailed breakdown of processing time for individual stages on mCBM data is provided in Appendix C.2.

5.6 Deployment and Evaluation in Production Environment

The preceding sections described the development and optimization of GPU-accelerated algorithms for STS reconstruction, demonstrating performance improvements in a controlled environment. To validate these results under real-world conditions, a complete production environment was established for deployment and testing. This section examines the transition from algorithm development to practical deployment, focusing on three key aspects: the container-based deployment strategy, systematic evaluation through controlled data challenges, and performance in an actual beam experiment. The integration of the optimized reconstruction chain into the broader CBM online software framework proved essential for reliable operation under experimental conditions, where factors such as variable data rates, system stability, and integration with other detector systems introduce additional complexity beyond pure algorithmic performance.

5.6.1 Container Building and Deployment

The deployment of the online processing software uses a container-based approach, providing necessary isolation of dependencies and simplifying the deployment process. This containerization strategy was especially crucial as bare metal access to the Virgo cluster, administered by the GSI IT department, processing nodes was not available, making containers a requirement rather than an optional design choice.

The container building process for the CBM online software follows a three-stage approach illustrated in Figure 5.9. The first stage creates two base containers that contain all dependencies: a development container containing build tools and all dependencies, and a minimal runtime container with only the required runtime libraries. This runtime container has a size of only about 160 MiB while the development container currently has a size of 2.6 GiB.

The base containers are maintained in a separate continuous integration pipeline that triggers rebuilds when dependency versions change. Additionally, container images are rebuild once per week automatically to ensure that operating system packages remain up-to-date with security patches and bug fixes, independent of dependency updates. By maintaining separate base containers, dependency updates like operating system versions, ROCm [7] or FairSoft [35] can be managed independently of the application code. When a dependency requires updating, only the relevant base container needs rebuilding, after which application containers automatically inherit the changes in subsequent builds.

The actual build process compiles the code within the development container. The resulting binaries and required runtime files are then copied into a new container based on the minimal runtime image. This multi-stage build process is automated through continuous integration using Kaniko [45]. The automation enables fast turnaround times - from code changes being pushed to a deployable container being available typically takes less than 10 minutes.

5.6.2 Data Challenges

The performance of the online reconstruction chain was evaluated using archived data from two mCBM benchmark runs (2391 and 2488) from the 2022 data taking campaign. To emulate the data flow of the actual experiment, a distributed system consisting of replay nodes and processing nodes was established, as illustrated in Figure 5.10.

In preparation for synthetic runs, dubbed ”Data Challenges”, the timeslice files from the benchmark runs were first merged and then split into 80 separate streams, enabling controlled testing at both original and accelerated data rates. Additionally, four replay nodes were configured to continuously stream timeslice data to the processing infrastructure.

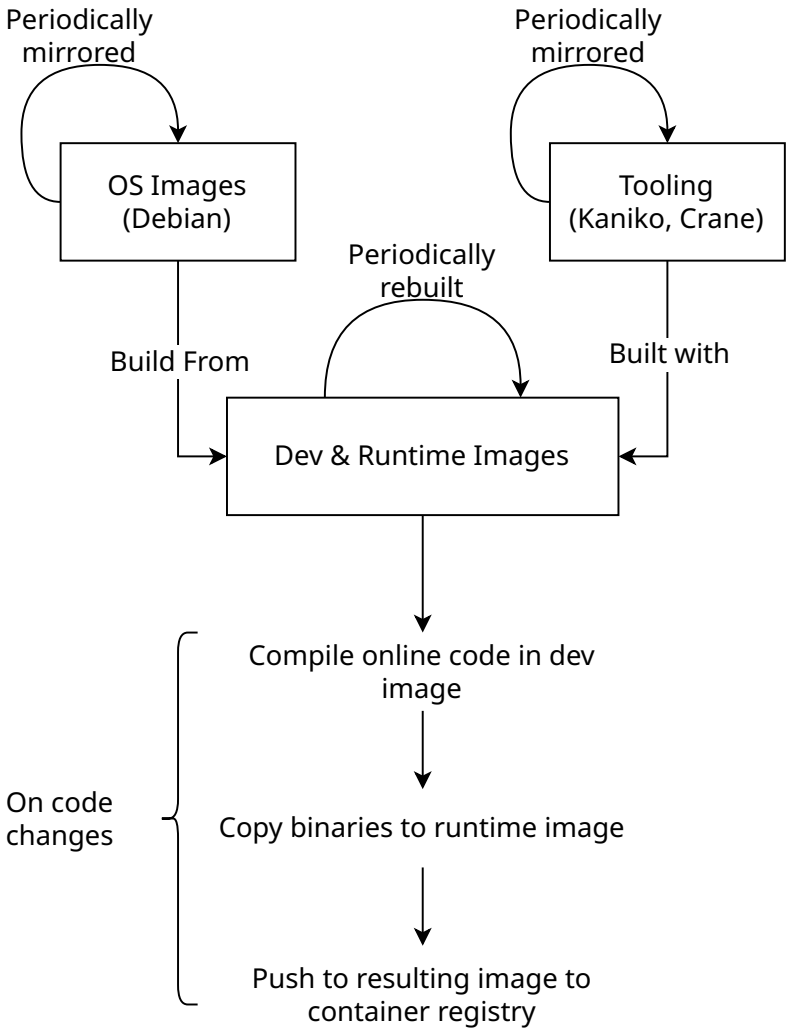

Figure 5.9: Flow chart of the container building process for CBM online software. Base images (OS and tooling) are periodically mirrored, from which the development and runtime images are built. Code changes trigger compilation using these images and deployment to the container registry deployment.

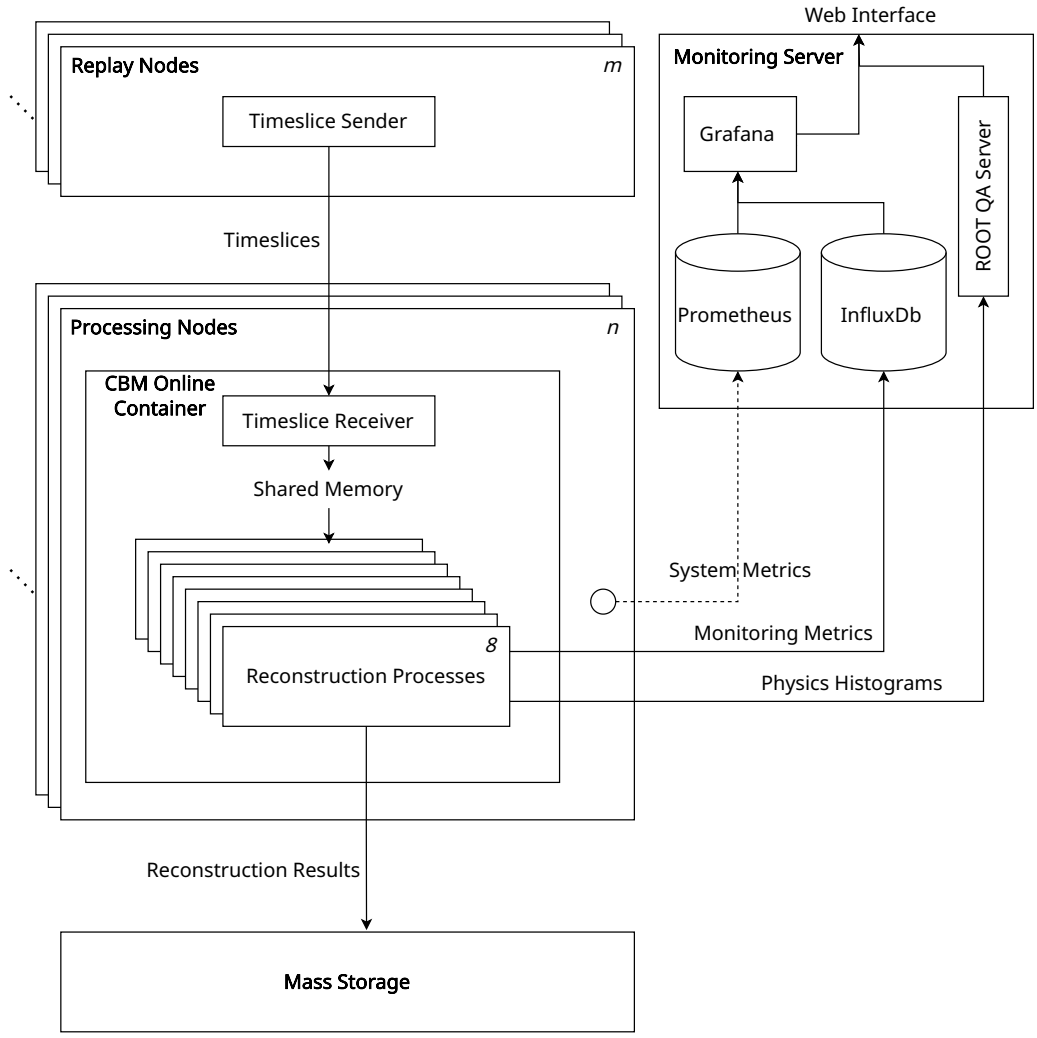

Figure 5.10: System setup of the mCBM Data Challenges.

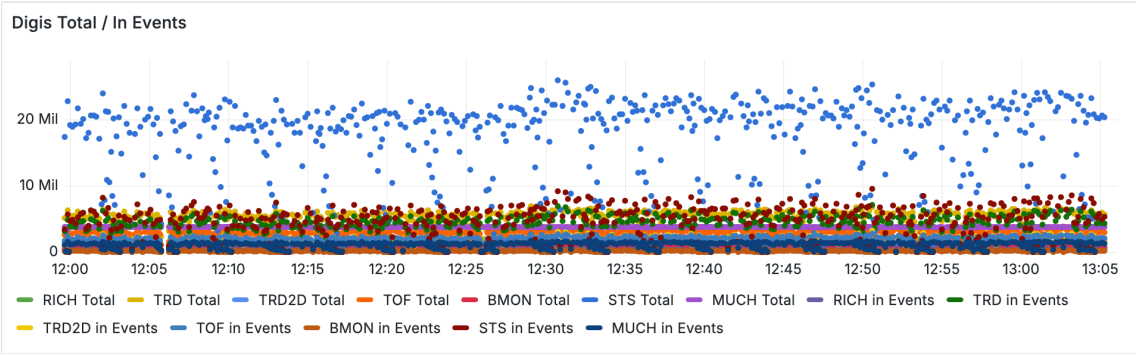

Figure 5.11: Number of digis per timeslice for every subsystem found during DC 4. Additionally the number of digis selected into events is shown.

The processing infrastructure consisted of 10 nodes from the Virgo computing cluster. Each processing node was configured to run 8 reconstruction processes in parallel, with jobs managed through the Slurm workload manager. The reconstruction software was deployed using the previously mentioned online containers.

Timeslices were streamed from the replay nodes to the processing nodes, where a TS Receiver process wrote the incoming data to shared memory. The reconstruction processes on each node read their input from this shared memory buffer, processed the data, and wrote the reconstruction results to Lustre mass storage.

A monitoring infrastructure was implemented to track system performance and data quality. The monitoring system consisted of three main components: Prometheus for system metrics collection, InfluxDB to store application level metrics, and Grafana [24] for visualization. Three categories of metrics were collected: system metrics tracking hardware utilization via the Prometheus node exporter [6], monitoring metrics capturing processing rates and performance[7], and physics histograms for quality assurance. A ROOT QA server was deployed to enable live inspection of the physics histograms.

This setup enabled systematic testing of the online reconstruction chain, starting with processing at the original data rate and gradually increasing to higher rates. The system's performance could be tested up to approximately 8 times the original data rate.

5.6.3 Results

The container-based deployment strategy and distributed testing environment described in the previous section enabled systematic evaluation of the CBM online chain, includ-

[6] The node exporter was provided by the GSI IT department and was running separately from the other components of the data challenge.

[7] Besides processing times for each timeframe, this includes for example values like the number of digis per subsystem per timeframe or the number of reconstructed tracks.

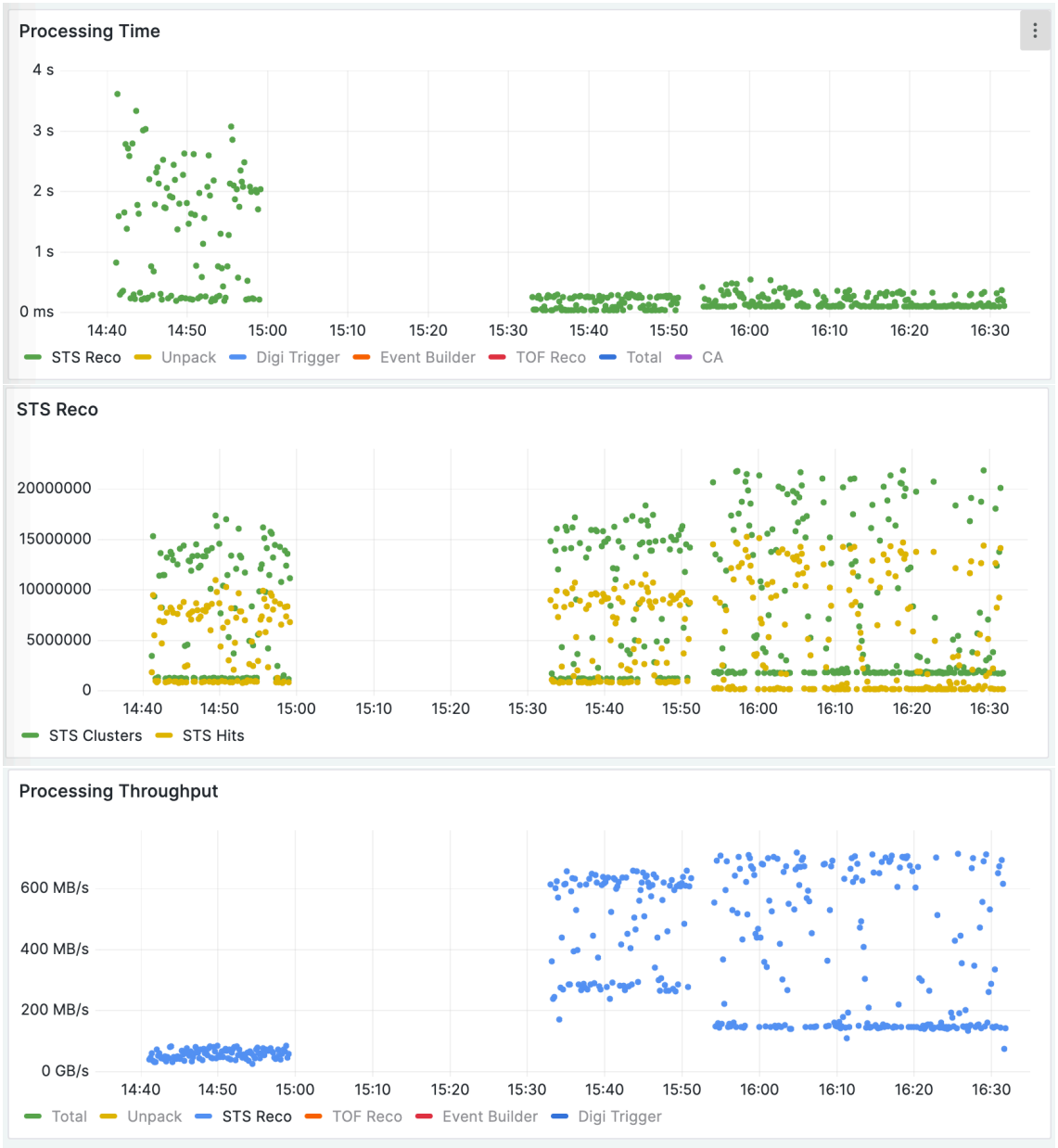

Figure 5.12: Performance metrics for STS reconstruction during a data challenge. The top panel shows processing time per timeslice, the middle panel displays processing throughput, and the bottom panel illustrates reconstruction quality through counts of STS clusters and hits. The transition from CPU to GPU processing at 15:30 demonstrates the significant performance improvement achieved by GPU acceleration.

ing the GPU-accelerated STS reconstruction, under conditions closely resembling the production environment. Through a series of progressively more complex data challenges, the system's performance, stability, and integration with other detector subsystems were assessed. This section presents the key findings from these tests, focusing particularly on processing throughput, resource utilization, and reconstruction quality.

Figure 5.11 shows the number of digis processed per timeslice during the fourth data challenge, demonstrating stable system operation over more than an hour. The STS detector contributed the largest data volume with approximately 20 M digis per timeslice, while other detectors showed consistent rates in the range of 2 M to 5 M digis. The significant reduction between total digis and digis selected into events, visible in the difference between solid and dashed lines, demonstrates the effectiveness of the event selection process.

The processing infrastructure successfully handled the distributed workload, with 80 parallel reconstruction processes spread across 10 nodes (8 processes per node). Real-time monitoring through the Grafana interface confirmed stable operation throughout the test period. The system not only processed data at the original recording rate but demonstrated scalability up to 8 times acceleration, validating its capability to handle higher data rates than currently required.

GPU acceleration was tested separately on a smaller number of nodes with MI100 GPUs. As illustrated in Figure 5.12, the transition from CPU to GPU processing at approximately 15:30 resulted in a dramatic reduction in processing time. While CPU processing exhibited considerable variability with times ranging from 0.5 s to 3.5 s per timeslice, the GPU-accelerated version consistently achieved processing times of approximately 0.12 s per timeslice. This up to 96 % reduction in execution time directly corresponds to the substantial improvement in processing throughput visible in the middle panel, where rates increased from less than 100 MB/s to over 600 MB/s in many cases.

The bottom panel shows that this performance improvement was achieved while maintaining reconstruction quality, as evidenced by the consistent relationship between the number of clusters and reconstructed hits across both processing modes. The reconstruction successfully identified between 15—20 million clusters and 5—10 million hits regardless of the processing architecture. The gap visible at approximately 15:50 represents the transition between the two benchmark data runs used in the data challenges.

Through these data challenges, the online reconstruction chain demonstrated stable operation over extended periods, successful integration of multiple detector systems, and efficient utilization of both CPU and GPU resources. The ability to process data at accelerated rates provides confidence in the system's readiness for deployment.

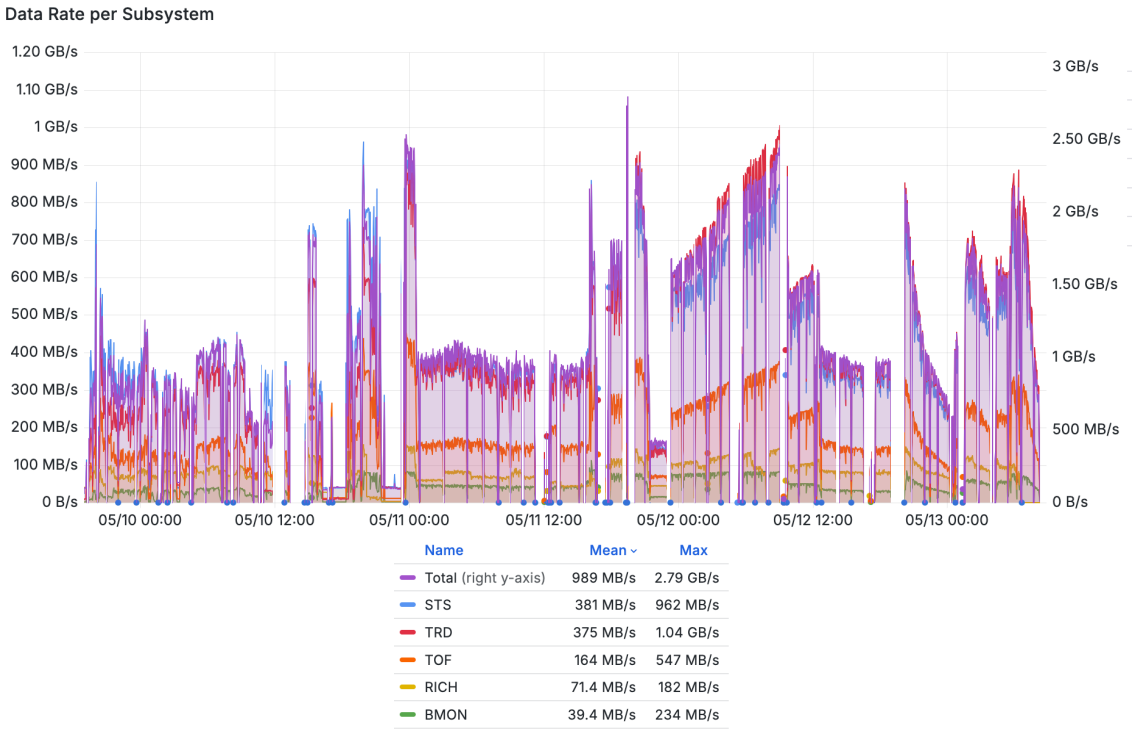

Figure 5.13: The produced datarates at mCBM beamtime.

5.6.4 May 2024 mCBM Beamtime

The developments of the online software system and experience from data challenges culminated in its first production deployment during the May 2024 mCBM beamtime. For four days, the online reconstruction ran continuously in parallel to recording of raw timeslices to disk. This ensured unprocessed data was still available for later offline analysis while marking the first demonstration of online processing capabilities in CBM.

The deployment required several technical adaptations. The unpacker implementations needed updates to handle changes in the firmware data format implemented since the 2022 benchmark runs. Additionally, the system had to be integrated with an updated readout setup and detector geometry. Over the duration of the beamtime new detector calibrations became available that had to be updated as well.

The system demonstrated robust performance processing continuous data streams. As shown in Figure 5.13, the average data rate reached approximately 800 MB/s, with peaks up to 2.4 GB/s. The STS detector contributed the largest portion at an average of 310 MB/s, reaching peaks of 900 MB/s. The periodic structure visible in the data rates reflects the spill pattern of the SIS18 synchrotron extraction.

Figure 5.14 shows the reconstruction performance for STS data over a three-hour pe-

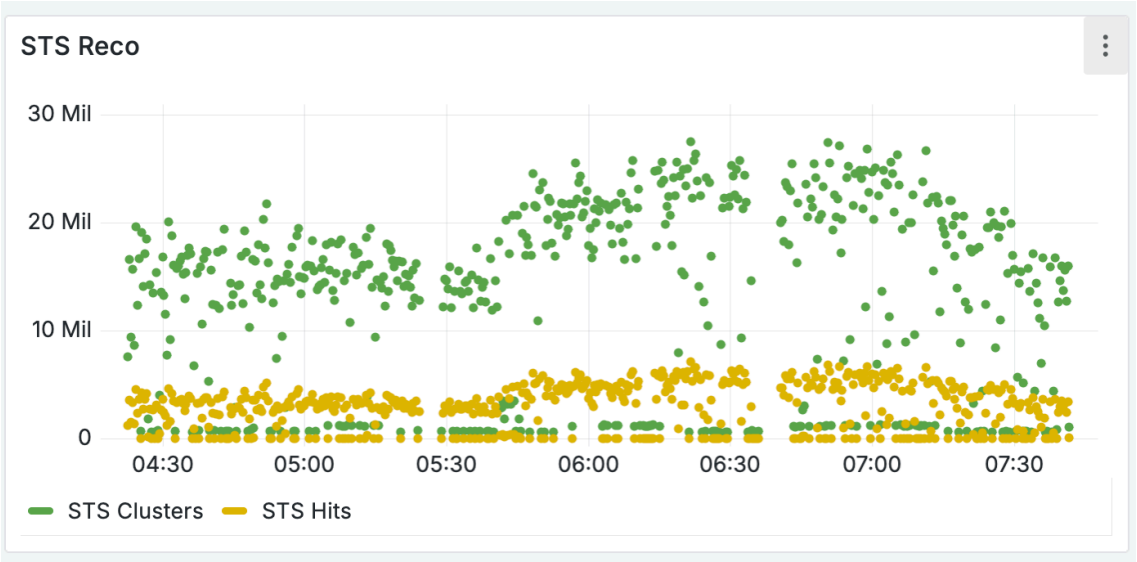

Figure 5.14: Monitoring data of the STS reconstruction from the mCBM beamtime showing the number of reconstructed clusters and hits per timeslice.

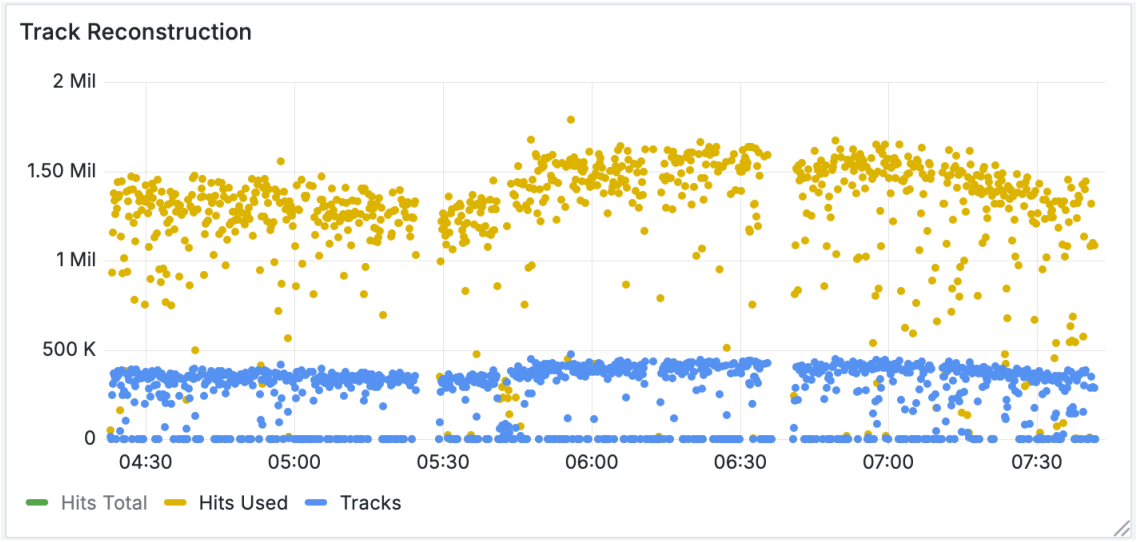

Figure 5.15: Monitoring data of the track reconstrcution from the mCBM beamtime showing the number of reconstructed tracks and hits attached to tracks per timeslice. The shown data is from the same time period as Figure 5.14.

riod. The system consistently processed between 3 and 4 million hits per timeframe, derived from 15–20 million clusters. The periodic structure visible in both figures reflects the spill pattern of the SIS18 synchrotron extraction, with intervals of high activity followed by pauses where minimal data was recorded. During these pauses these some clusters are still found in the recorded noise, but only a negligle number of hits and tracks are created. These reconstructed hits provided the foundation for track finding, with Figure 5.15 demonstrating stable track reconstruction over the same time period. During active spills, approximately 1.5×10^6 hits per timeframe contributed to the reconstruction of around 500×10^3 tracks, maintaining consistent track yields throughout the beam delivery cycles.

The successful deployment during the May 2024 mCBM beamtime demonstrated that the reconstruction chain could operate effectively under real-world conditions, processing data from actual detectors rather than idealized simulations. The system maintained stable performance despite varying beam conditions, including fluctuating data rates corresponding to the SIS18 extraction cycle, detector noise, and evolving calibration parameters. This operational validation provided crucial evidence that the reconstruction algorithms are robust enough to handle the unpredictable aspects of experimental data taking, including hardware imperfections and beam instabilities that cannot be fully captured in simulations. The continuous operation over four days further confirmed the system's reliability for extended production runs, establishing a solid foundation for the future CBM experiment's online processing infrastructure.

Chapter 6

Conclusion

This dissertation presents several contributions to GPU computing in high-energy physics experiments, focusing on three main areas.

For the ALICE experiment, several components of the TPC reconstruction chain were optimized for GPU processing. This includes zero-suppression decoders for two formats, which use warp-level ballot operations for parallel scan replacing multiple register shuffles with a single instruction. The track merger was parallelized using graph connected components to identify independent sets, enabling a 30x speedup without requiring synchronization or locks. Cluster gathering to host memory at the end of the processing chain was improved by using shared memory buffering and coalesced 128-byte DMA transfers. This kernel is about three orders of magnitude faster than performing naive device-to-host transfers. The presented software pieces have been successfully deployed in the ALICE online processing since the start of the LHC Run 3 in 2022.

The second contribution introduced xpu, a lightweight C++ library for portable GPU programming that enables writing code for CUDA, HIP, and SYCL backends without sacrificing performance. Compared to similar libraries, xpu supports using vendor-optimized compilers for each backend simultaneously through a custom symbol resolution system that handles C++ name mangling across different compilers. The library provides buffer classes for RAII-based memory management while maintaining explicit control over allocations, and includes a built-in profiling API for measuring kernel execution and DMA transfer times. Through separate compilation of device code and dynamic backend selection, xpu achieves performance within 0.2 % of native implementations. Tests of the STS unpacker showed Intel oneAPI being 35-50 % slower, while AdaptiveCpp matched xpu's performance on NVIDIA hardware but was 47 % slower on AMD GPUs.

Finally, for the CBM experiment, a complete GPU-accelerated reconstruction chain for the Silicon Tracking System was developed. The implementation achieved a 122x speedup compared to the existing CPU-based reconstruction through several key optimizations. A custom merge sort implementation outperforms both NVIDIA's CUB and AMD's hipCUB libraries by 10 % on NVIDIA hardware and up to 38 % on AMD GPUs

when processing the full detector configuration. Cluster finding was parallelized using atomic operations on 32 bit values that encode both cluster membership and predecessor flags, enabling lock-free updates when thousands of threads process digis simultaneously. Even before GPU acceleration, the optimized data structures and algorithms developed for the port provided over a 2x speedup in single-threaded CPU execution. The online reconstruction chain was successfully deployed in production at the May 2024 mCBM beamtime, processing data rates up to 2.4 GB/s in real-time over the duration of four days.

Several ongoing development efforts are building on these contributions. In CBM, work continues to extend GPU processing capabilities ahead of the planned 2028 data taking. A prototype GPU implementation of the STS unpacker has been developed by S. Heinemann [42], while G. Kozlov is working on a port of the track reconstruction to GPUs [46]. Both implementations use the xpu library. Additionally, the MUCH group is investigating a GPU implementation of their hitfinder [62]. Integration of the online processing with CBM's simulation framework remains an important goal. This will enable validation with Monte Carlo data and testing of the full detector configuration before hardware deployment. Development of raw data simulation capabilities will furthermore allow synthetic data runs as a critical step towards processing live data.

Development in ALICE also focuses on expanding the scope of GPU processing. Work is underway to extend the current TPC tracking to full barrel tracking [56], moving toward complete global track reconstruction on GPUs. Additionally, research by C. Sonnabend into machine learning approaches for TPC clustering is ongoing [63]. This could replace the current TPC clusterizer, touched upon in Chapter 3, providing a higher level of noise suppression and furthermore predict particle momentum to aid in tracking, which is not possible with the current clusterizer.

While the xpu library provides the core functionality for accelerator programming, it offers several directions for enhancement. Support for more advanced features like CUDA graphs would enable more sophisticated execution patterns. Emulation of GPU-blocks on the CPU-side via coroutines could allow for easier debugging of device code and elimination of special cases, similar to hipCPU [72]. Integration of vectorization support through `std::simd` [26] on CPUs and corresponding block- or warp-wide operations on GPUs would further close the gap for efficient code between CPU architectures and GPUs.

These software developments occur against a backdrop of evolving hardware capabilities. Modern CPU architectures, particularly with AVX-512 vector instructions, could present a viable alternative to GPU processing for certain workloads. The emergence of cost-effective ARM-based processors adds another dimension to hardware choices.

As experiments evolve and data rates increase, efficient processing becomes ever more critical. The achievements in both ALICE and CBM demonstrate that careful algorithmic optimization combined with GPU acceleration can yield order-of-magnitude performance improvements. The real-time processing capabilities demonstrated through

production deployments in both experiments not only validate this approach but establish a foundation for meeting future computational demands in high-energy physics experiments.

Appendix A

xpu Examples

A.1 Vector Add

This section presents a complete example program demonstrating vector addition using xpu. The example shows how to:

- Define and implement a GPU kernel
- Manage device memory allocation and transfers
- Launch kernels and synchronize execution
- Set up a CMake build for multiple GPU backends

First, let's look at the kernel declaration in `vector_add.h`:

```
#pragma once
#include <xpu/device.h>

// Dummy type to identify the device library
struct DeviceLib {};

struct VectorAdd : xpu::kernel<DeviceLib> {
    using block_size = xpu::block_size<256>; // Set block size to 256 threads
    using context = xpu::kernel_context<>; // Shorthand for context type

    XPU_D void operator()(
        context& ctx,
        xpu::buffer<const float> a,
        xpu::buffer<const float> b,
        xpu::buffer<float> c,
        size_t N);
};
```

The kernel is implemented as a callable object inheriting from `xpu::kernel`. We specify a block size of 256 threads through the `block_size` type. The kernel takes three buffers as input: two for the source vectors (`a` and `b`) and one for the output vector (`c`). The size parameter `N` specifies the length of the vectors.

The implementation in `vector_add.cpp`:

```
#include "vector_add.h"

XPU_IMAGE(DeviceLib);
XPU_EXPORT(VectorAdd);

XPU_D void VectorAdd::operator()(
    context& ctx,
    xpu::buffer<const float> a,
    xpu::buffer<const float> b,
    xpu::buffer<float> c,
    size_t N)
{
    // Calculate global thread index
    int idx = ctx.block_idx_x() * ctx.block_dim_x() + ctx.thread_idx_x();
    if (idx >= N) return;

    // Perform vector addition
    c[idx] = a[idx] + b[idx];
}
```

Each thread computes its global index using the block and thread indices from the context. If the index exceeds the vector size, the thread returns early. Otherwise, it performs the addition for its assigned element.

The host code in `main.cpp` demonstrates memory management and kernel execution:

```cpp
#include "vector_add.h"
#include <xpu/host.h>
#include <iostream>

int main() {
    xpu::initialize(); // Initialize xpu runtime

    constexpr size_t N = 1'000'000; // Problem size

    // Allocate input/output buffers
    xpu::buffer<float> a{N, xpu::buf_io};
    xpu::buffer<float> b{N, xpu::buf_io};
    xpu::buffer<float> c{N, xpu::buf_io};

    // Initialize input data
    xpu::h_view<float> ah{a}, bh{b};
    for (size_t i = 0; i < N; i++) {
        ah[i] = static_cast<float>(i);
        bh[i] = static_cast<float>(i * 2);
    }

    xpu::queue queue;   // Create command queue and execute kernel

    queue.copy(a, xpu::h2d); // Copy input data to device
    queue.copy(b, xpu::h2d);

    // Launch kernel with N threads total
    queue.launch<VectorAdd>(xpu::n_threads(N), a, b, c, N);

    queue.copy(c, xpu::d2h);     // Copy results back to host

    queue.wait();     // Wait for all operations to complete

    // Verify results
    bool success = true;
    xpu::h_view<float> ch{c};
    for (size_t i = 0; i < N; i++) {
        float expected = i + (i * 2);
        success &= (ch[i] == expected);
    }

    std::cout << (success ? "Success!" : "Failed!") << std::endl;
    return success ? 0 : 1;
}
```

Finally, this is the CMake configuration to build the example with support for multiple GPU backends:

```
cmake_minimum_required(VERSION 3.11)
project(xpu-vector-add)

# Enable desired backends
option(XPU_ENABLE_CUDA "Enable CUDA backend" ON)
option(XPU_ENABLE_HIP "Enable HIP backend" ON)
option(XPU_ENABLE_SYCL "Enable SYCL backend" OFF)

# Add xpu as a subdirectory or using FetchContent
include(FetchContent)
FetchContent_Declare(xpu
    GIT_REPOSITORY https://github.com/fweig/xpu
    GIT_TAG master
)
FetchContent_MakeAvailable(xpu)

# Create executable
add_executable(vector_add main.cpp vector_add.cpp)
target_link_libraries(vector_add PRIVATE xpu)

# Compile device code for enabled backends
xpu_attach(vector_add vector_add.cpp)
```

To build and run the example on the first available CUDA GPU:

```
cmake -S . -B build
cd build
make
XPU_DEVICE=cuda0 ./vector_add
```

Appendix B

ALICE Zero Suppression Decoding CPU Performance

B.1 Without Thread Pinning

Figure B.1 and Figure B.2 show the performance of the row-based and link-based decoder without thread affinity. This is analog to Figure 3.6 in Section 3.4.3 where the performance of the dense-link based decoder is discussed.

Figures B.3, B.4 and B.5 show the performance for all three decoding algorithms with the AMD EPYC 7552 CPU.

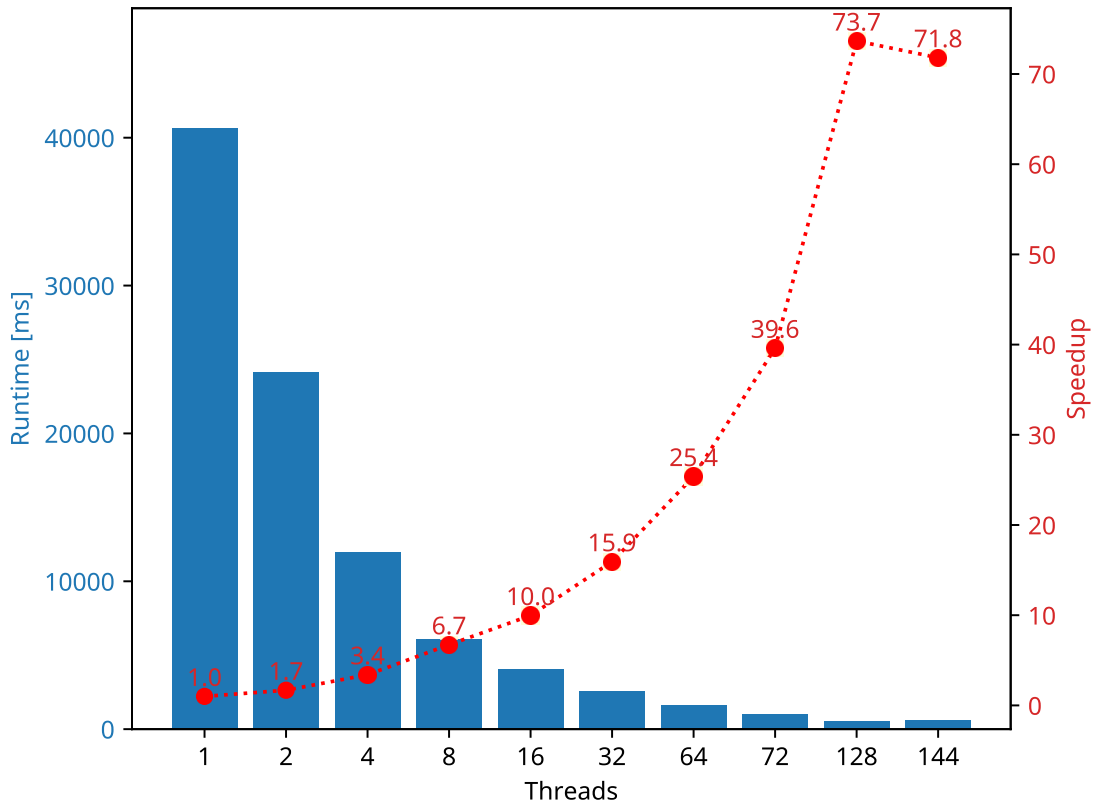

Figure B.1: CPU - ZS Row based. (without thread affinity)

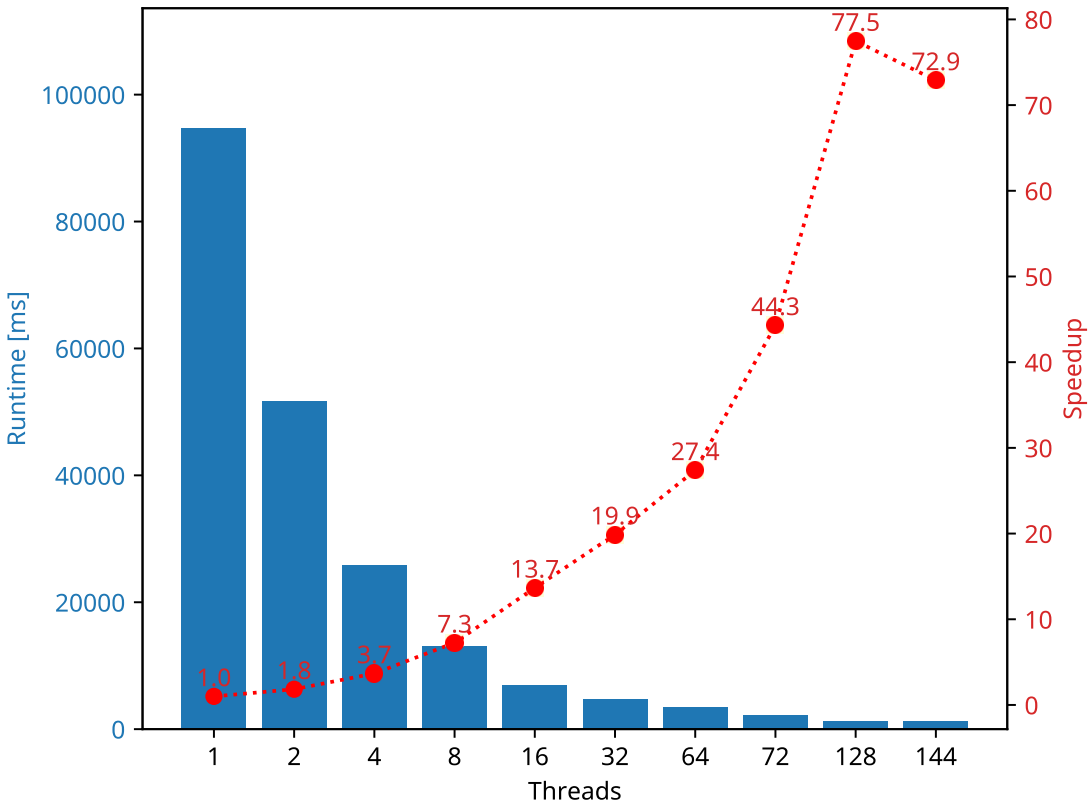

Figure B.2: CPU - ZS Link based. (without thread affinity)

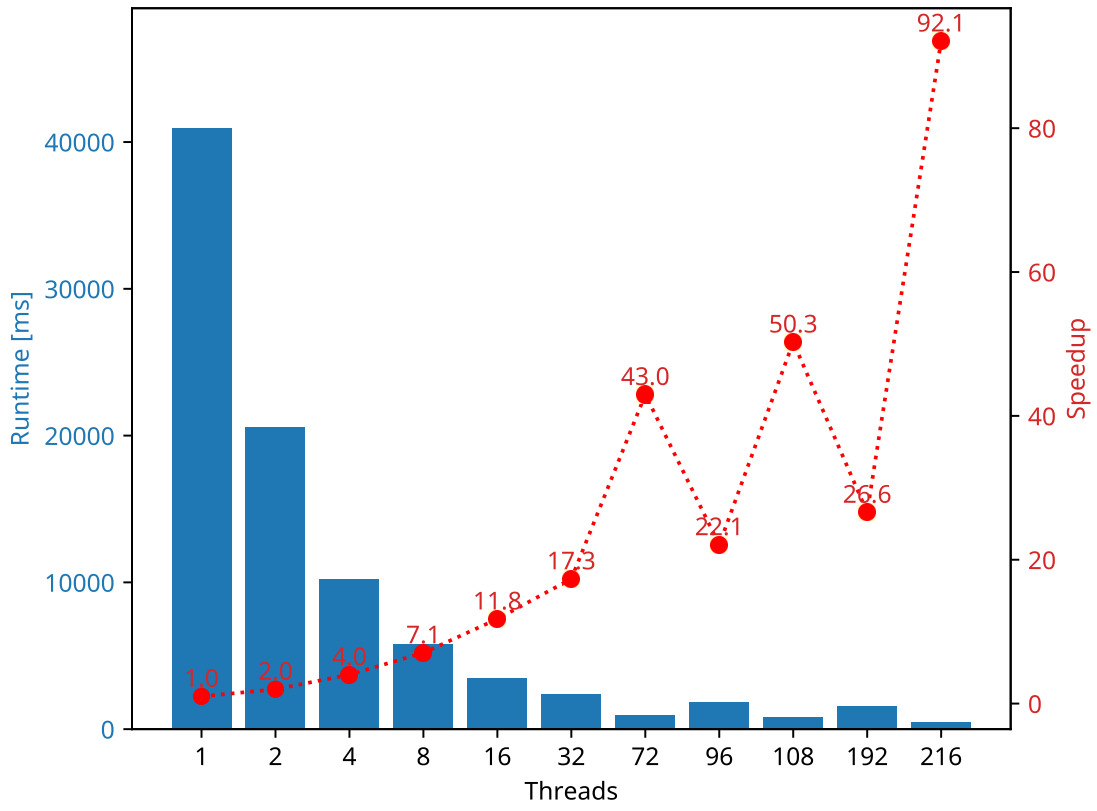

Figure B.3: CPU - ZS Row based. (without thread affinity)

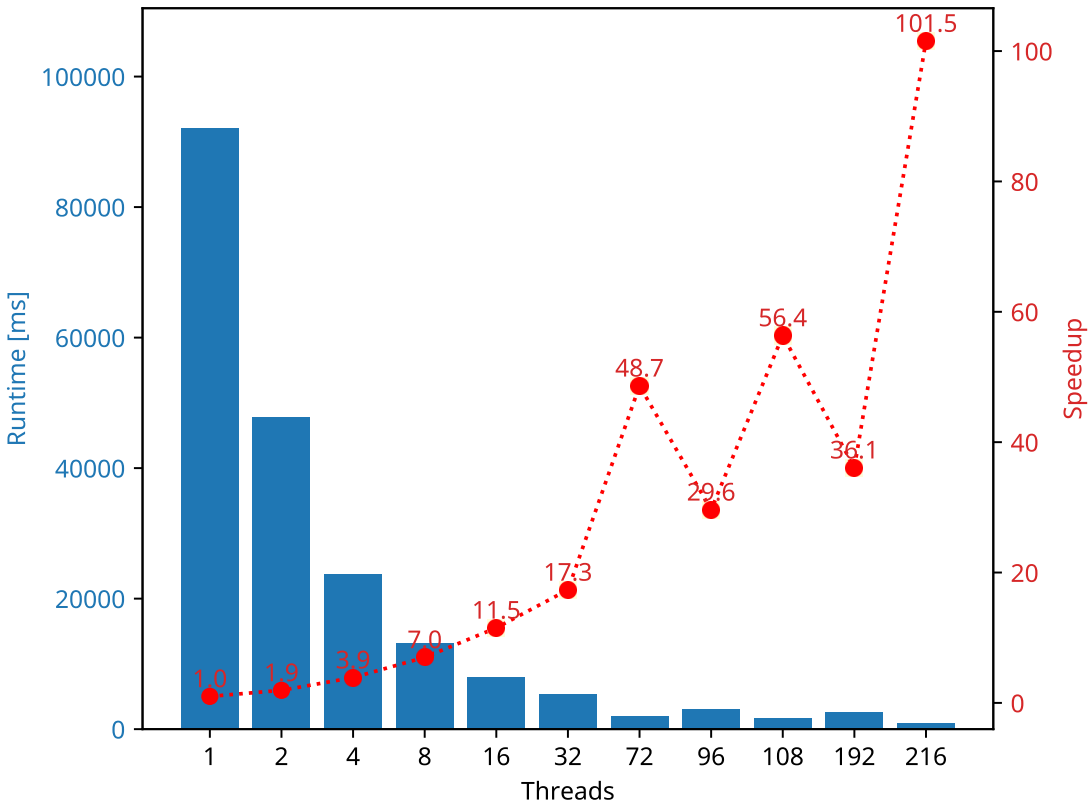

Figure B.4: CPU - ZS Row based. (without thread affinity)

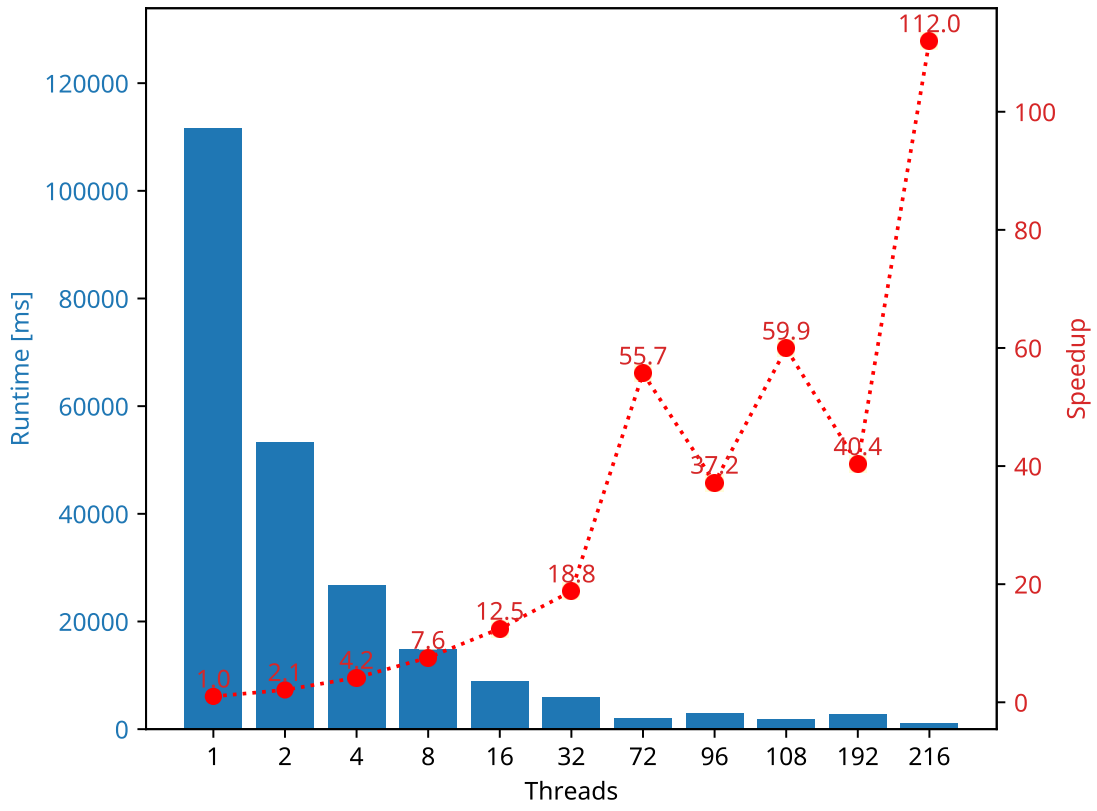

Figure B.5: CPU - ZS Row based. (without thread affinity)

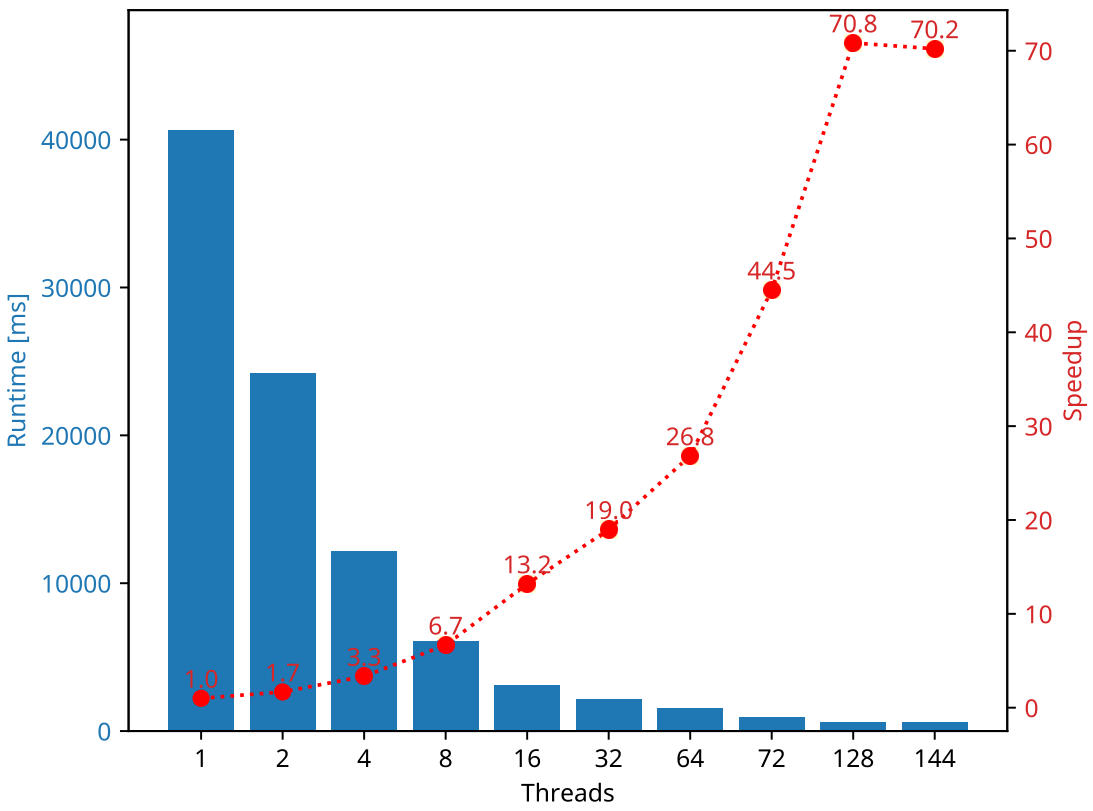

Figure B.6: CPU - ZS Row based. (with socket pinning)

B.2 With Socket Pinning

Figure B.6 and Figure B.7 show the performance of the row-based and link-based decoder without thread affinity. This is analog to Figure 3.7 in Section 3.4.3.

Futhermore Figure B.8, Figure B.9 and Figure B.10 show the performance of the row-based, the link-based decoder and the dense-link-based decoder with socket pinning on the AMD EPYC 7552 CPU used in MI100 nodes.

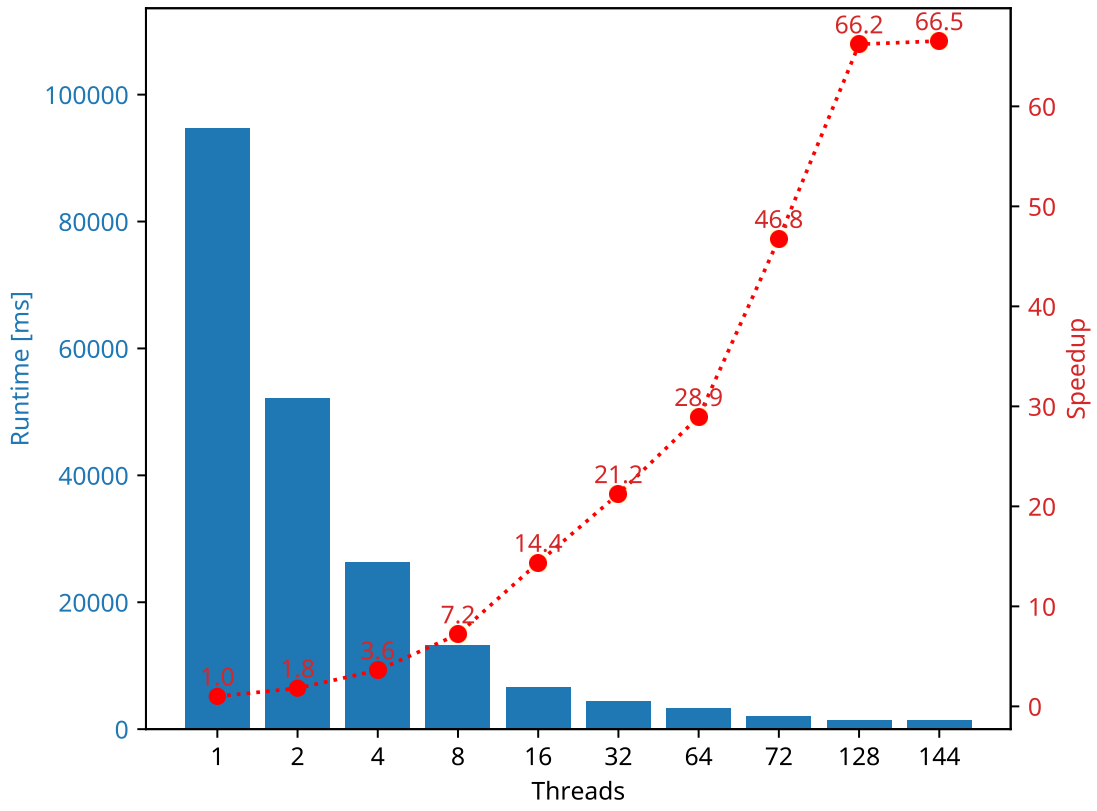

Figure B.7: CPU - ZS Link based. (with socket pinning)

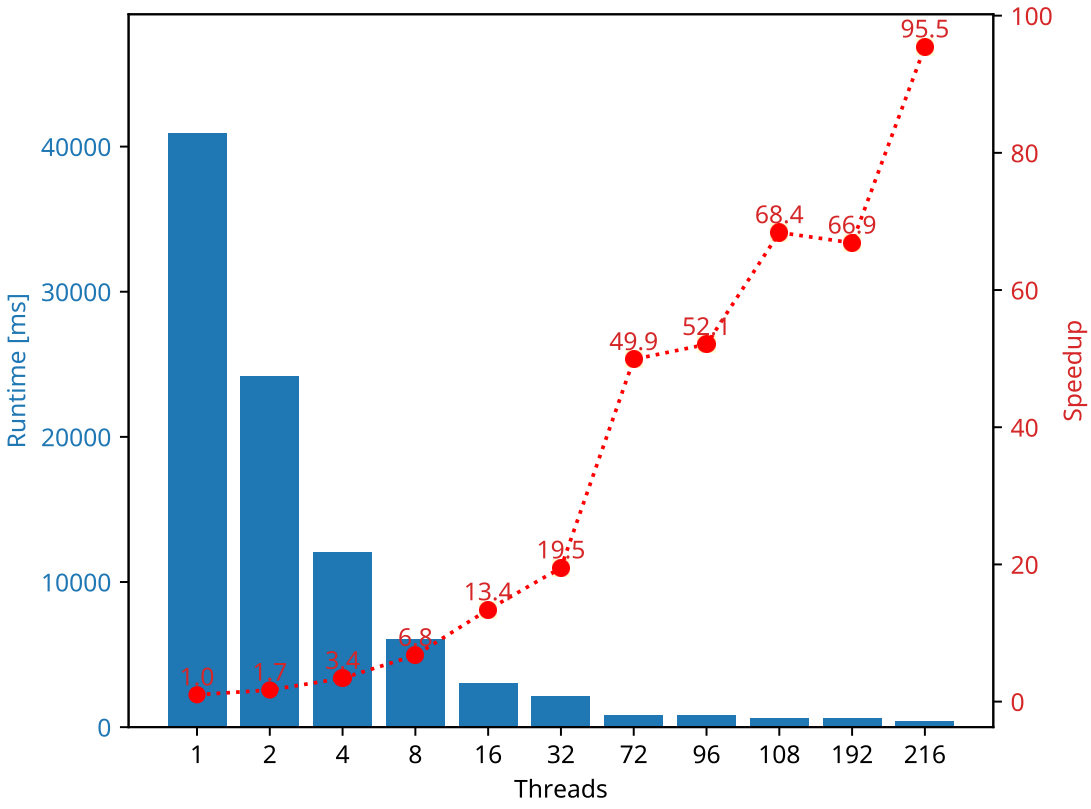

Figure B.8: CPU - ZS Row based on EPYC 7552. (with socket pinning)

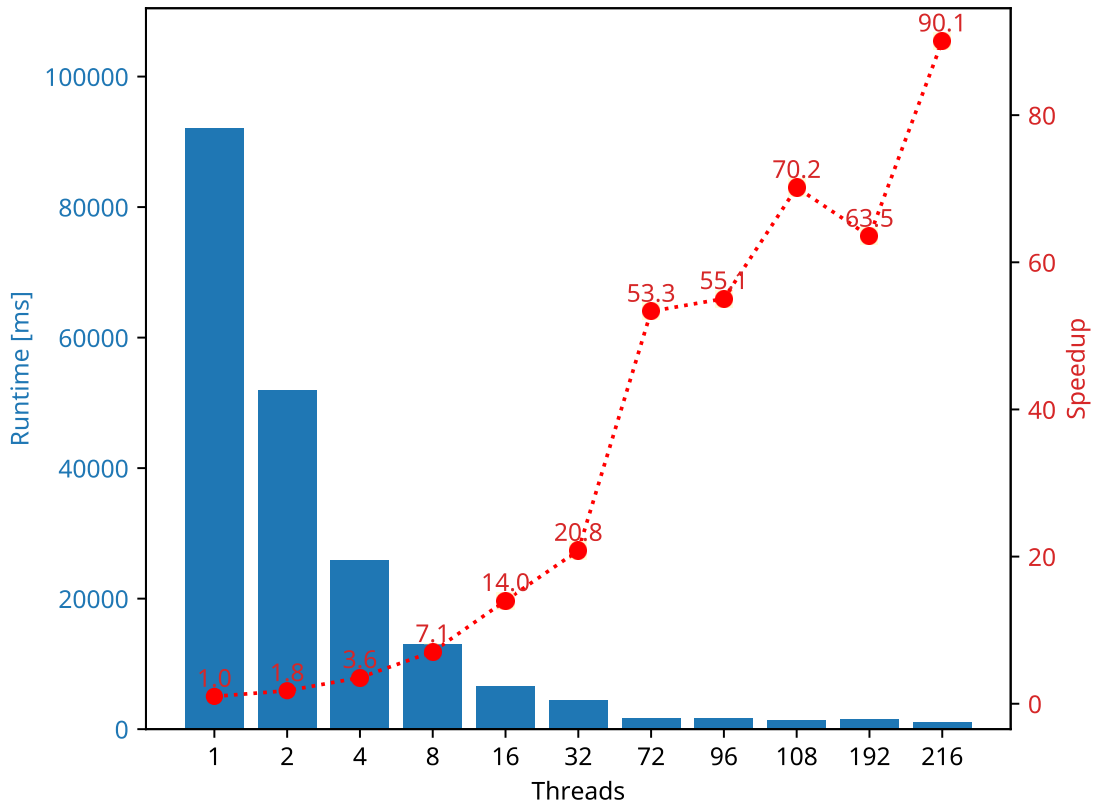

Figure B.9: CPU - ZS Link based on EPYC 7552. (with socket pinning)

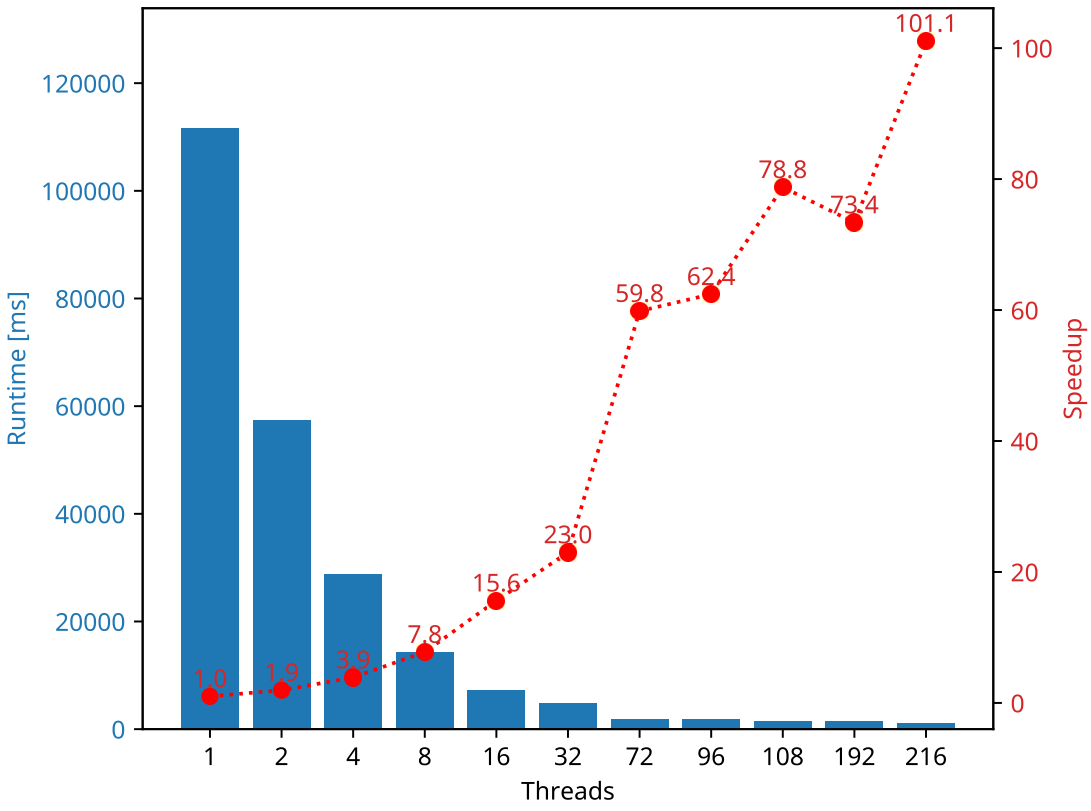

Figure B.10: CPU - ZS Link based on EPYC 7552. (with socket pinning)

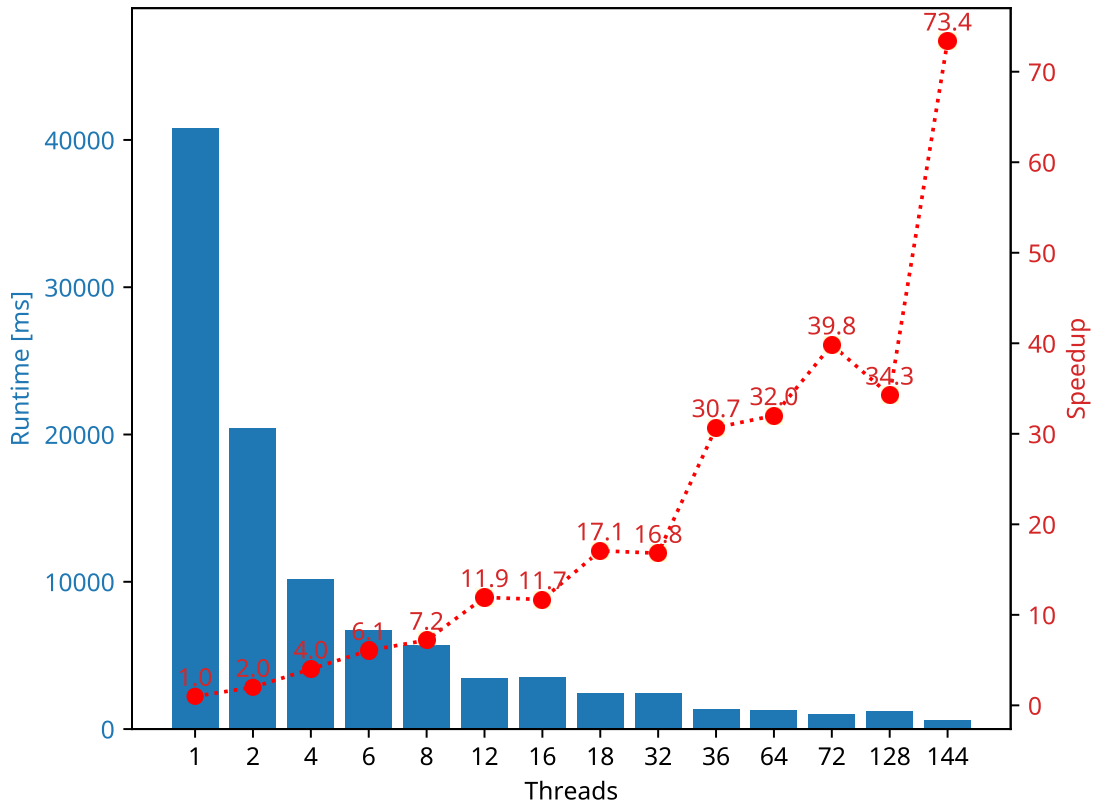

Figure B.11: CPU - ZS Row based. (with per die pinning)

B.3 With Die Pinning

Figure B.11, Figure B.12 and Figure B.13 show the performance of the row-based, the link-based decoder and the dense-link-based decoder when pinning threads to individual dies of the EPYC 7452. Each die contains four physical cores and a L3 cache shared by the four cores [66]. This approach improves performance in general for lower thread counts, but slows down execution when the entire CPU is occupied.

Futhermore Figure B.14, Figure B.14 and Figure B.16 show the performance of the row-based, the link-based decoder and the dense-link-based decoder with die pinning on the AMD EPYC 7552 CPU used in MI100 nodes.

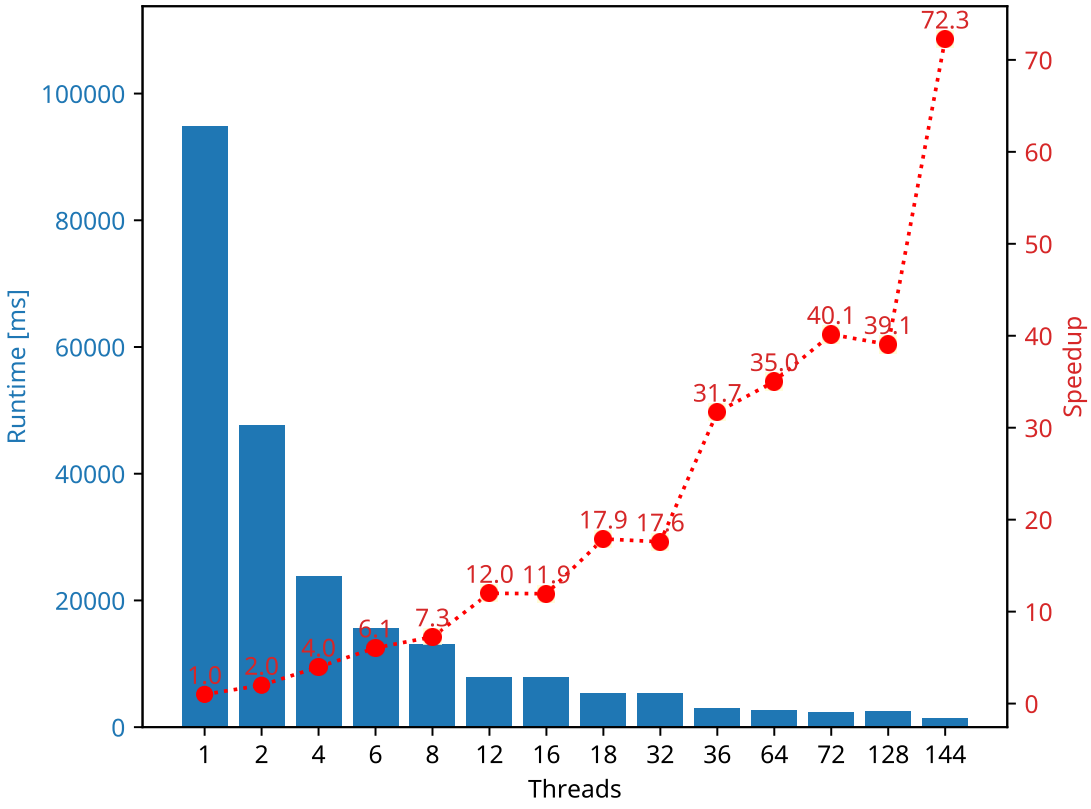

Figure B.12: CPU - ZS Link based. (with per die pinning)

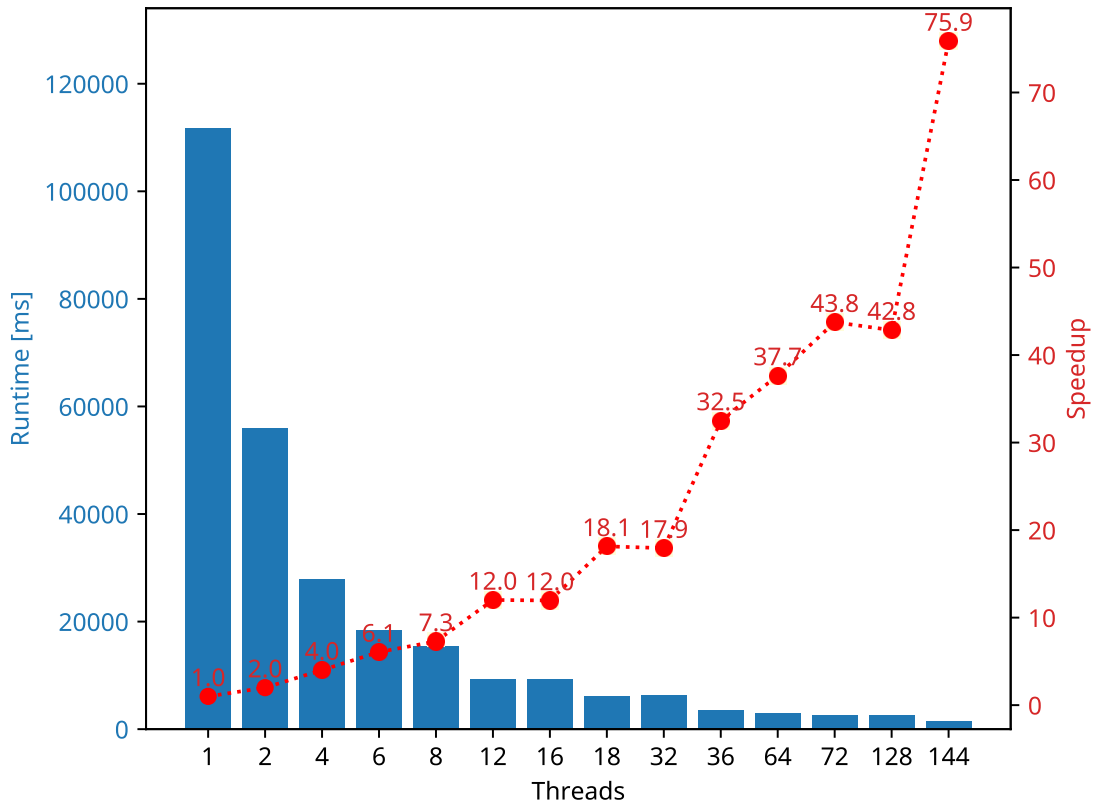

Figure B.13: CPU - ZS Dense link based. (with per die pinning)

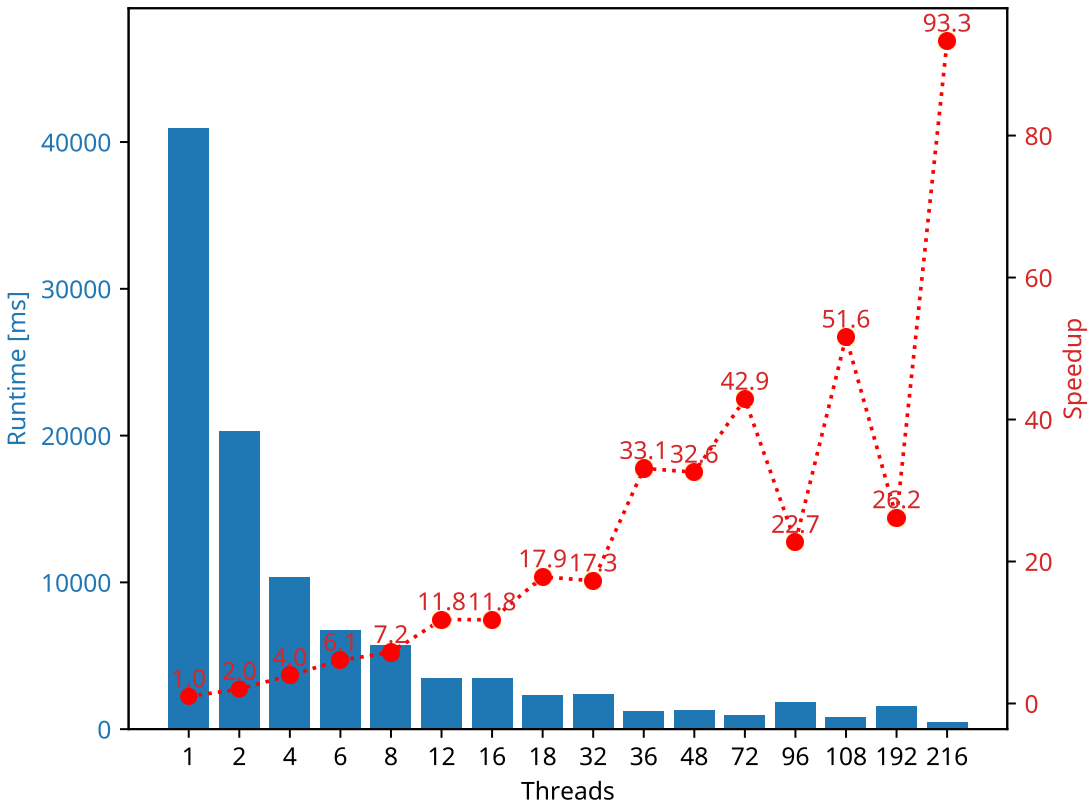

Figure B.14: CPU - Row Based ZS on EPYC 7552. (with per die pinning)

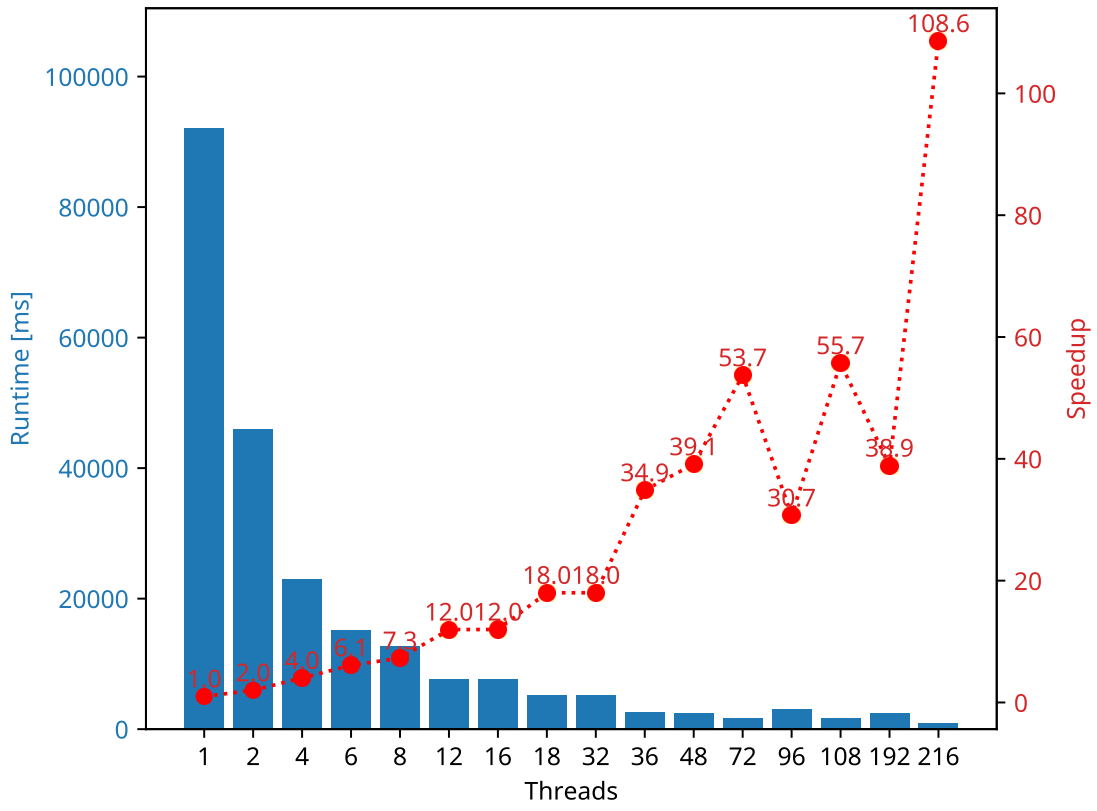

Figure B.15: CPU - Link Based ZS on EPYC 7552. (with per die pinning)

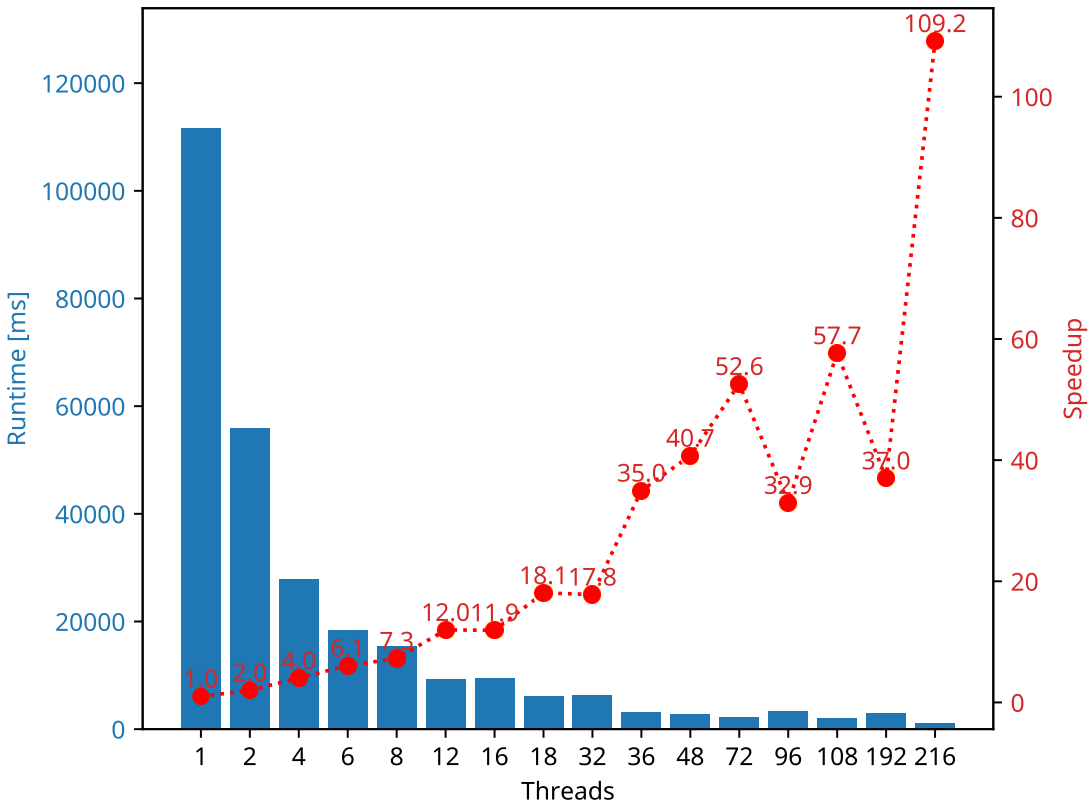

Figure B.16: CPU - Dense Link Based ZS on EPYC 7552. (with per die pinning)

Appendix C

STS Hitfinder Substeps Performance

C.1 Performance on CBM Data

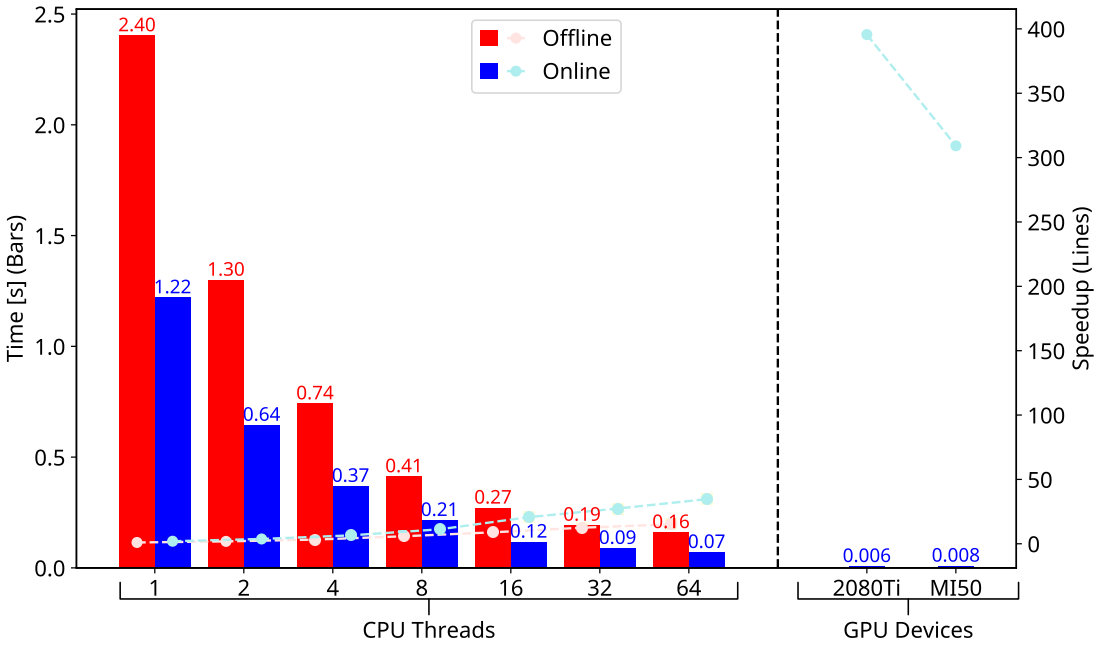

Figure C.1: Performance comparison of the offline and online STS sorting stages.

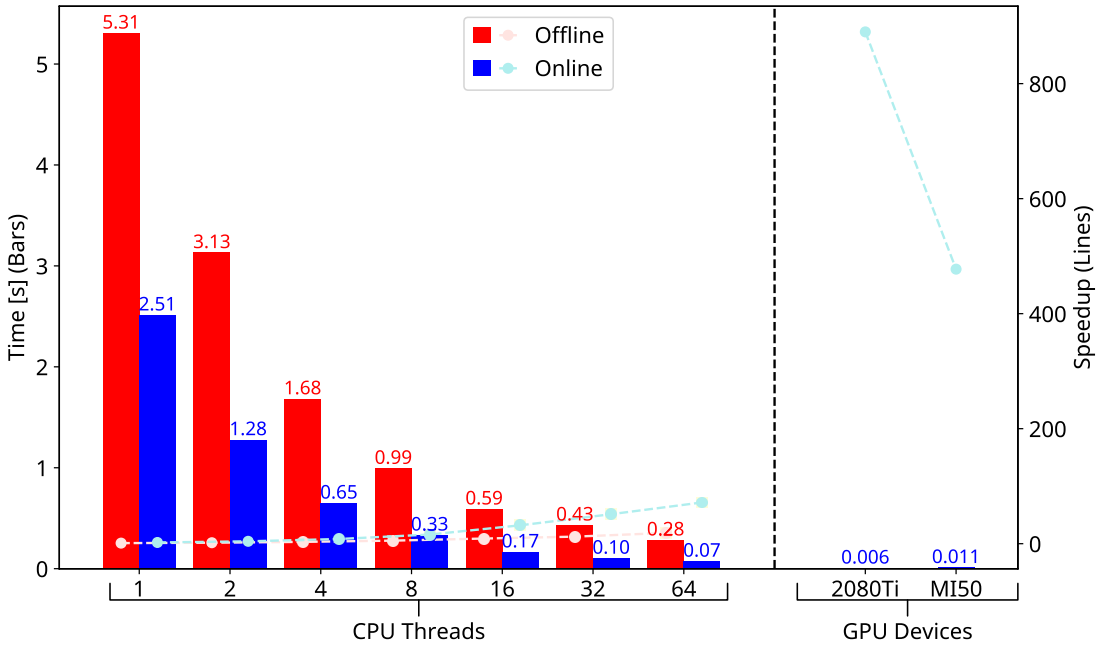

Figure C.2: Performance comparison of the offline and online STS clustering.

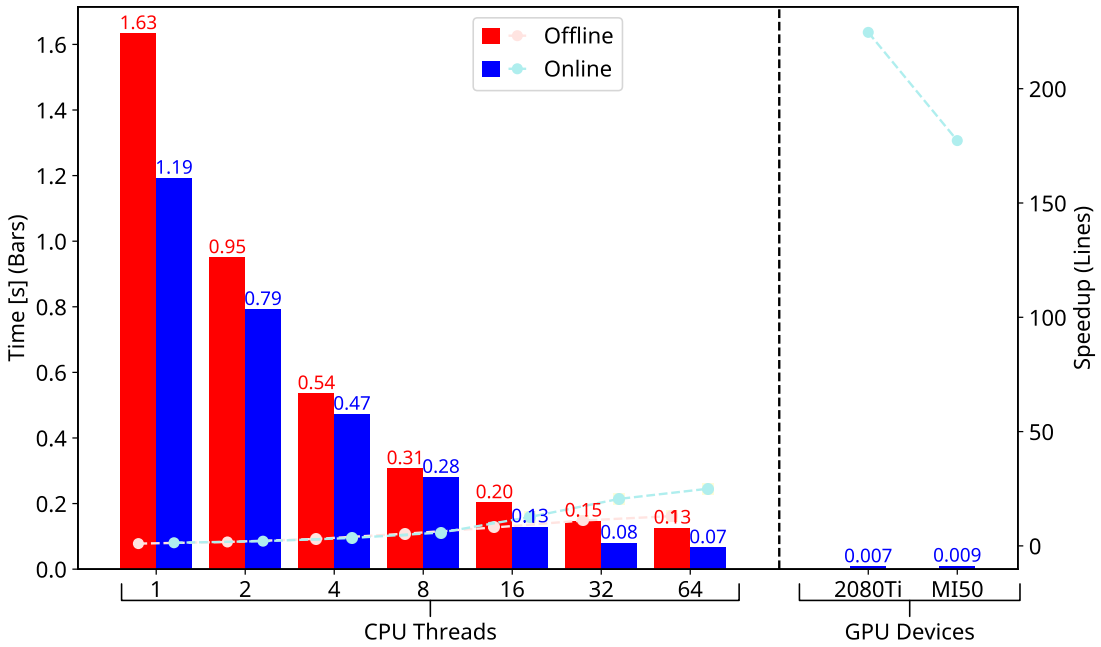

Figure C.3: Performance comparison of the offline and online STS hit creation.

C.2 Performance on mCBM Data

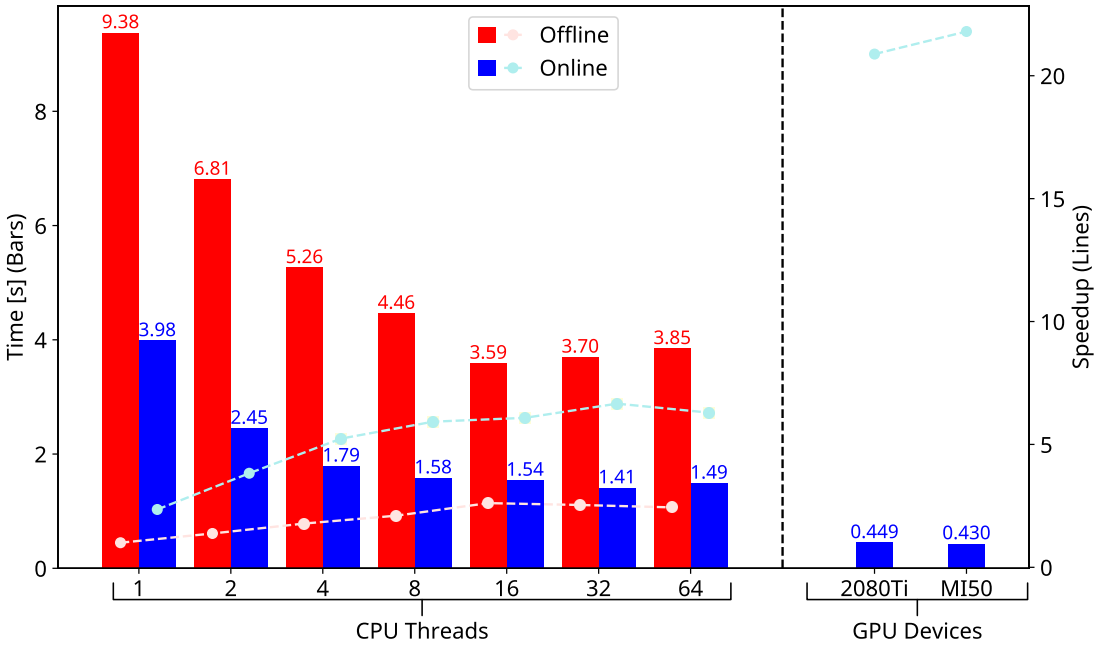

Figure C.4: Performance comparison of the offline and online STS sorting on mCBM data.

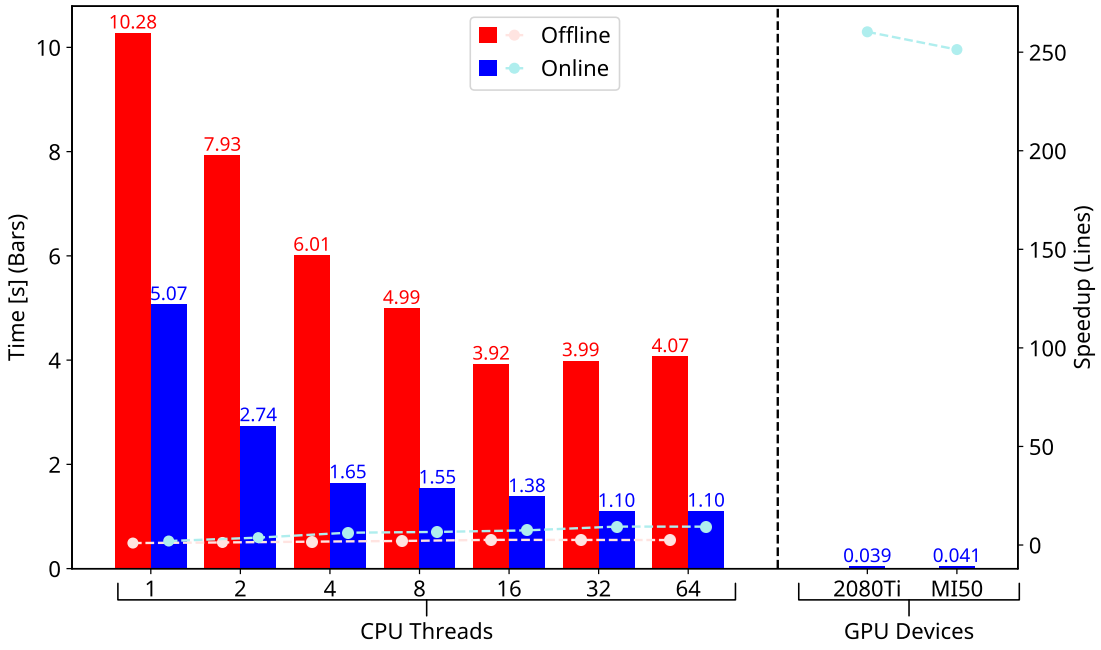

Figure C.5: Performance comparison of the offline and online STS cluster finding stage on mCBM data.

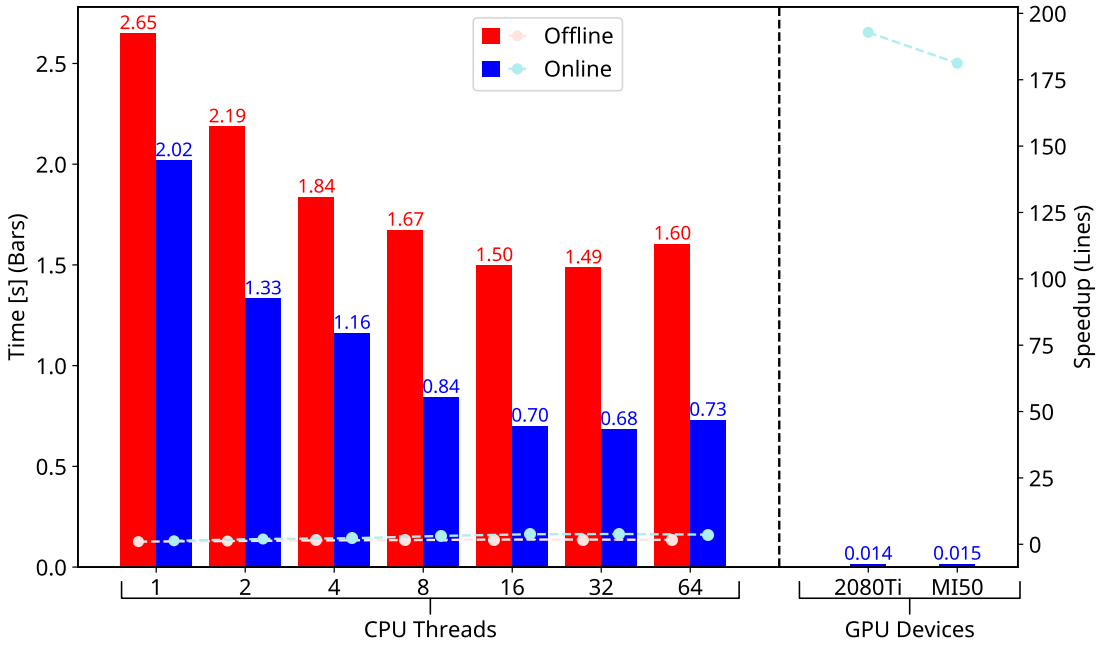

Figure C.6: Performance comparison of the offline and online STS hit finding stage on mCBM data.

Bibliography

[1] CBM Collaboration. *The Transition Radiation Detector of the CBM Experiment at FAIR : Technical Design Report for the CBM Transition Radiation Detector (TRD)*. Tech. rep. FAIR Technical Design Report. Darmstadt: Facility for Antiproton and Ion Research, 2018, 165 p. DOI: 10.15120/GSI-2018-01091. URL: https://repository.gsi.de/record/217478.

[2] Jan de Cuveland et al. *Technical Design Report for the CBM Online Systems – Part I, DAQ and FLES Entry Stage*. Tech. rep. -. FAIR Technical Design Report; ccby4. Darmstadt, 2023. DOI: 10.15120/GSI-2023-00739. URL: https://repository.gsi.de/record/340597.

[3] Johann Heuser et al., eds. *[GSI Report 2013-4] Technical Design Report for the CBM Silicon Tracking System (STS)*. Darmstadt: GSI, 2013, 167 p. URL: https://repository.gsi.de/record/54798.

[4] R. Aaij et al. "Allen: A High-Level Trigger on GPUs for LHCb". In: *Computing and Software for Big Science* 4.1 (Apr. 2020). ISSN: 2510-2044. DOI: 10.1007/s41781-020-00039-7. URL: http://dx.doi.org/10.1007/s41781-020-00039-7.

[5] K. Aamodt et al. "The ALICE experiment at the CERN LHC". In: *JINST* 3 (2008), S08002. DOI: 10.1088/1748-0221/3/08/S08002.

[6] Shreyasi Acharya et al. "ALICE upgrades during the LHC Long Shutdown 2". In: *JINST* 19.05 (2024), P05062. DOI: 10.1088/1748-0221/19/05/P05062. arXiv: 2302.01238 [physics.ins-det].

[7] Advanced Micro Devices. *ROCm - Open Source Platform for HPC and Machine Learning*. https://www.amd.com/en/products/software/rocm.html. Open-source software platform for GPU computing [Accessed: 2024-10-31]. 2024.

[8] Advanced Micro Devices, Inc. *HIP Programming Guide*. https://rocm.docs.amd.com/projects/HIP/en/latest/index.html. [Accessed: 2024-10-31].

[9] ALICE Collaboration. "Real-time data processing in the ALICE High Level Trigger at the LHC". In: *Computer Physics Communications* 242 (2019), pp. 25–48. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2019.04.011. URL: https://www.sciencedirect.com/science/article/pii/S0010465519301250.

[10] *Schematics of the ALICE subdetectors*. https://commons.wikimedia.org/wiki/File:2012-Aug-02-ALICE_3D_v0_with_Text_(1)_2.jpg. General Photo. 2014.

[11] Tyler Allen and Rong Ge. “In-depth analyses of unified virtual memory system for GPU accelerated computing”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3480855. URL: https://doi.org/10.1145/3458817.3480855.

[12] J. Alme et al. “The ALICE TPC, a large 3-dimensional tracking device with fast readout for ultra-high multiplicity events”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 622.1 (Oct. 2010), pp. 316–367. ISSN: 0168-9002. DOI: 10.1016/j.nima.2010.04.042. URL: http://dx.doi.org/10.1016/j.nima.2010.04.042.

[13] Christian Bierlich et al. *A comprehensive guide to the physics and usage of PYTHIA 8.3*. 2022. arXiv: 2203.11601 [hep-ph]. URL: https://arxiv.org/abs/2203.11601.

[14] Florian Boeck. *Optimierung der lokalen Rekonstruktion des STS-Detektors am CBM-Experiment*. Bachelor’s Thesis. Goethe University, 2019.

[15] Rene Brun and Fons Rademakers. “ROOT: An object oriented data analysis framework”. In: *Nucl. Instrum. Meth. A* 389 (1997), pp. 81–86. DOI: 10.1016/S0168-9002(97)00048-X.

[16] Michele Caini. *entt/core/type_info.hpp*. https://github.com/skypjack/entt/blob/4ed58d22305207a63890e7c8851db4500f1fee36/src/entt/core/type_info.hpp. [Accessed 12-11-2024].

[17] CBM Collaboration. *mCBM@SIS18 - A CBM full system test-setup for high-rate nucleus-nucleus collisions at GSI / FAIR*. CBM. Darmstadt: GSI, 2017, 58 S. DOI: 10.15120/GSI-2019-00977. URL: https://repository.gsi.de/record/220072.

[18] *Definition of CbmHit*. https://git.cbm.gsi.de/computing/cbmroot/-/blob/024b7bb99c6edb6e15e209a00c44900d477612fa/core/data/CbmHit.h. [Accessed 24-01-2025].

[19] *Definition of CbmPixelHit*. https://git.cbm.gsi.de/computing/cbmroot/-/blob/024b7bb99c6edb6e15e209a00c44900d477612fa/core/data/CbmPixelHit.h. [Accessed 24-01-2025].

[20] *Definition of CbmStsHit*. https://git.cbm.gsi.de/computing/cbmroot/-/blob/024b7bb99c6edb6e15e209a00c44900d477612fa/core/data/sts/CbmStsHit.h. [Accessed 24-01-2025].

[21] *Definition of CbmCluster*. https://git.cbm.gsi.de/computing/cbmroot/-/blob/024b7bb99c6edb6e15e209a00c44900d477612fa/core/data/CbmCluster.h. [Accessed 24-01-2025].

[22] *Definition of CbmStsCluster*. https://git.cbm.gsi.de/computing/cbmroot/-/blob/024b7bb99c6edb6e15e209a00c44900d477612fa/core/data/sts/CbmStsCluster.h. [Accessed 24-01-2025].

[23] *Definition of CbmStsDigi*. https://git.cbm.gsi.de/computing/cbmroot/-/blob/024b7bb99c6edb6e15e209a00c44900d477612fa/core/data/sts/CbmStsDigi.h. [Accessed 24-01-2025].

[24] Mainak Chakraborty and Ajit Pratap Kundan. "Grafana". In: *Monitoring Cloud-Native Applications: Lead Agile Operations Confidently Using Open Source Software*. Berkeley, CA: Apress, 2021, pp. 187–240. ISBN: 978-1-4842-6888-9. DOI: 10.1007/978-1-4842-6888-9_6. URL: https://doi.org/10.1007/978-1-4842-6888-9_6.

[25] The ALICE Collaboration. *Upgrade of the ALICE Time Projection Chamber*. Tech. rep. CERN, 2013.

[26] *Documentation of the C++ SIMD standard library*. https://en.cppreference.com/w/cpp/experimental/simd. [Accessed 07-02-2025].

[27] *Documentation of cub::BlockRadixSort*. https://nvidia.github.io/cccl/cub/api/classcub_1_1BlockRadixSort.html. [Accessed 29-01-2025].

[28] *Documentation of cub::DeviceSegmentedSort*. https://nvidia.github.io/cccl/cub/api/structcub_1_1DeviceSegmentedSort.html. [Accessed 29-01-2025].

[29] J. de Cuveland et al. "Accelerating Online Software Development with CBM Data Challenges". In: *CBM Progress Report 2023*. Ed. by Volker Friese, Ilya Selyuzhenkov, and Piotr Gasik. GSI Helmholtzzentrum für Schwerionenforschung, 2024, pp. 133–134. ISBN: 978-3-9822127-2-2. DOI: 10.15120/GSI-2024-00765. URL: https://repository.gsi.de/record/352779.

[30] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[31] Joshua H. Davis et al. *Taking GPU Programming Models to Task for Performance Portability*. 2024. arXiv: 2402.08950 [cs.DC]. URL: https://arxiv.org/abs/2402.08950.

[32] I. Deppner and N. Herrmann. "The CBM Time-of-Flight system". In: *Journal of Instrumentation* 14.09 (Sept. 2019), pp. C09020–C09020. ISSN: 1748-0221. DOI: 10.1088/1748-0221/14/09/c09020. URL: http://dx.doi.org/10.1088/1748-0221/14/09/C09020.

[33] Eulisse, Giulio and Rohr, David. "The O2 software framework and GPU usage in ALICE online and offline reconstruction in Run 3". In: *EPJ Web of Conf.* 295 (2024), p. 05022. DOI: 10.1051/epjconf/202429505022. URL: https://doi.org/10.1051/epjconf/202429505022.

[34] Evgeny Lavrik on behalf of the CBM Collaboration. "Compressed Baryonic Matter experiment at FAIR". In: *AIP Conference Proceedings* 2163.1 (Oct. 2019), p. 030009. ISSN: 0094-243X. DOI: 10.1063/1.5130095. eprint: https://pubs.aip.org/aip/acp/article-pdf/doi/10.1063/1.5130095/14196649/030009_1_online.pdf. URL: https://doi.org/10.1063/1.5130095.

[35] FairRoot Group at GSI. *FairSoft - Repository for installation routines of the external software required by FairRoot*. https://github.com/FairRootGroup/FairSoft. [Accessed 01-02-2025].

[36] Volker Friese and Jan de Cuveland. "Towards an online-capable software stack". In: *CBM Progress Report 2021*. Ed. by Peter Senger and Volker Friese. Darmstadt: GSI Darmstadt, 2022, pp. 173–174. ISBN: 978-3-9822127-0-8. DOI: 10.15120/GSI-2022-00599. URL: https://repository.gsi.de/record/246663.

[37] Friese, Volker. "A cluster-finding algorithm for free-streaming data". In: *EPJ Web Conf.* 214 (2019), p. 01008. DOI: 10.1051/epjconf/201921401008. URL: https://doi.org/10.1051/epjconf/201921401008.

[38] Oded Green, Saher Odeh, and Yitzhak Birk. *Merge Path - A Visually Intuitive Approach to Parallel Merging*. 2014. arXiv: 1406.2628 [cs.DC].

[39] Klaus Gross and Jürgen Eschke. "Facility for antiproton and ion research (FAIR)". In: *Nuclear Physics News* 16.1 (2006), pp. 5–8.

[40] The Khronos® SYCL™ Working Group. *SYCL 2020 Specification (revision 8)*. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html. [Accessed 15-08-2024].

[41] *GSI Helmholtzzentrum für Schwerionenforschung*. www.gsi.de/. [Accessed 11-07-2024].

[42] Sebastian Heinemann. *Analyse und Optimierung der parallelen Prozessierung des Unpackings, ausgehend vom STS-Detektor am mCBM-Experiment*. MA thesis. Goethe University, 2023.

[43] Advanced Micro Devices. *hipCUB - Reusable software components for ROCm developers*. https://github.com/ROCm/hipCUB. [Accessed 27-01-2025].

[44] Kilian Hunold. *Optimierung der lokalen Rekonstruktion beim STS-Detektor des CBM-Experiments zur effizienten Prozessierung auf GPUs*. MA thesis. Goethe University, 2022.

[45] Google. *Kaniko - Build Container Images In Kubernetes*. https://github.com/GoogleContainerTools/kaniko. [Accessed 01-02-2025].

[46] Grigory Kozlov. *Optimization of compute-intensive reconstruction tasks for efficient usage of CPUs and GPUs*. https://indico.desy.de/event/46129/contributions/174400/. FIDIUM collaboration meeting, RWTH Aachen. Sept. 2024.

[47] Matthias Kretz and Volker Lindenstruth. "Vc: A C++ library for explicit vectorization". In: *Softw. Pract. Exper.* 42.11 (Nov. 2012), pp. 1409–1430. ISSN: 0038-0644. DOI: 10.1002/spe.1149. URL: https://doi.org/10.1002/spe.1149.

[48] Hanna Malygina. *Hit reconstruction for the Silicon Tracking System of the CBM experiment*. PhD thesis. Goethe University, 2018.

[49] John Nickolls et al. "Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?" In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. URL: https://doi.org/10.1145/1365490.1365500.

[50] Nvida. *Nvidia CUB*. https://nvidia.github.io/cccl/cub/. [Accessed 06-02-2025].

[51] *Definitions and ODR (One Definition Rule) - cppreference.com*. https://en.cppreference.com/w/cpp/language/definition. [Accessed 10-12-2024].

[52] *Definition of PartitionedSpan*. https://git.cbm.gsi.de/computing/cbmroot/-/blob/024b7bb99c6edb6e15e209a00c44900d477612fa/algo/base/PartitionedSpan.h. [Accessed 24-01-2025].

[53] *Definition of PartitionedVector*. https://git.cbm.gsi.de/computing/cbmroot/-/blob/024b7bb99c6edb6e15e209a00c44900d477612fa/algo/base/PartitionedVector.h. [Accessed 24-01-2025].

[54] Saman Sedighi Rad. *Efficient sorting algorithms on GPUs for the CBM experiment - with focus on time sorting of STS measurements*. MA thesis. Goethe University, 2023.

[55] *RAII / Resource Acquisition Is Initialization- cppreference.com*. https://en.cppreference.com/w/cpp/language/raii. [Accessed 10-12-2024].

[56] David Rohr. "Usage of GPUs for online and offline reconstruction in ALICE in Run 3". In: *PoS* ICHEP2024 (2024), p. 1012. DOI: 10.22323/1.476.1012.

[57] Federico Ronchetti. *ALICE O2: GPU pictures for CERN Courier article (Sep-Oct 2023)*. https://cds.cern.ch/record/2866675. General Photo. 2023.

[58] Federico Ronchetti et al. *Efficient high performance computing with the ALICE Event Processing Nodes GPU-based farm*. 2024. arXiv: 2412.13755 [hep-ex]. URL: https://arxiv.org/abs/2412.13755.

[59] Alessandro Scarabotto. *The LHCb Trigger in Run 3*. https://indi.to/gwv4h. Bochum, Germany, Sept. 2024.

[60] P Senger. "Probing dense QCD matter in the laboratory—The CBM experiment at FAIR". In: *Physica Scripta* 95.7 (May 2020), p. 074003. ISSN: 1402-4896. DOI: 10.1088/1402-4896/ab8c14. URL: http://dx.doi.org/10.1088/1402-4896/ab8c14.

[61] *STS — gsi.de*. https://www.gsi.de/work/forschung/cbmnqm/cbm/activities/sts. [Accessed 24-07-2024].

[62] Vikas Singhal. *MUCH software status*. https://indico.gsi.de/event/19018/contributions/78547/. 43rd CBM Collaboration Meeting, GSI, Darmstadt. Mar. 2024.

[63] Christian Sonnabend. *Neural network clusterization for the ALICE TPC online computing*. https://indico.cern.ch/event/1338689/contributions/6015422/. Conference on Computing in High Energy and Nuclear Physics, Krakow. Oct. 2024.

[64] Jens Stadlmann et al. "SIS18 operation and recent development". In: *JACoW* IPAC (2023), TUPA163. DOI: 10.18429/JACoW-IPAC2023-TUPA163.

[65] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science & Engineering* 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[66] David Suggs, Mahesh Subramony, and Dan Bouvier. “The AMD ”Zen 2” Processor”. In: *IEEE Micro* PP (Feb. 2020), pp. 1–1. DOI: 10.1109/MM.2020.2974217.

[67] Michael Sutton et al. *Adaptive Work-Efficient Connected Components on the GPU*. 2016. arXiv: 1612.01178 [cs.DC].

[68] *ROOT - TClonesArray Class Reference*. https://root.cern.ch/doc/master/classTClonesArray.html. [Accessed 27-01-2025].

[69] *ROOT - TObject Class Reference*. https://root.cern.ch/doc/master/classTObject.html. [Accessed 27-01-2025].

[70] Christian R. Trott et al. “Kokkos 3: Programming Model Extensions for the Exascale Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[71] Mohammad Al-Turany et al. “Extending the FairRoot framework to allow for simulation and reconstruction of free streaming data”. In: *Journal of Physics: Conference Series*. Vol. 513. IOP Publishing. 2014, p. 022001.

[72] Alex Voicu. *GitHub - ROCm/HIP-CPU: An implementation of HIP that works on CPUs, across OSes. — github.com*. https://github.com/ROCm/HIP-CPU. [Accessed 31-10-2024].

[73] F. Weiglhofer. “Online STS hitfinder on GPUs”. In: *CBM Progress Report 2023*. Ed. by Volker Friese, Ilya Selyuzhenkov, and Piotr Gasik. GSI Helmholtzzentrum für Schwerionenforschung, 2024, p. 139. ISBN: 978-3-9822127-2-2. DOI: 10.15120/GSI-2024-00765. URL: https://repository.gsi.de/record/352779.

[74] Felix Weiglhofer. *Optimierung des Cluster Finders auf GPUs im TPC-Detektor bei ALICE*. MA thesis. Goethe University, 2019.

[75] Wojciech Zabolotny. *Status of STS firmware*. https://indico.gsi.de/event/19534/contributions/82441/. [Accessed 24-01-2025]. Sept. 2024.

[76] Erik Zenker et al. “Alpaka – An Abstraction Library for Parallel Kernel Acceleration”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 631–640. DOI: 10.1109/IPDPSW.2016.50.

Zusammenfassung

Einleitung

Moderne Hochenergiephysik-Experimente erzeugen Datenmengen in bisher ungekanntem Ausmaß, was effiziente Verarbeitungslösungen in Echtzeit erfordert. Das ALICE-Experiment am CERN und das geplante CBM-Experiment an der FAIR-Anlage stehen vor der Herausforderung, Datenströme von mehr als 1 TB/s zu verarbeiten. Diese Dissertation befasst sich mit der Entwicklung und Implementierung von GPU-beschleunigten Algorithmen für eine Echtzeitverarbeitung solcher Datenmengen.

Im ALICE-Experiment wurde für Run 3 ein kontinuierliches Auslesemodell eingeführt, bei dem Daten von Schwerionenkollisionen mit bis zu 50 kHz ohne Hardware-Trigger verarbeitet werden. Das CBM-Experiment plant einen ähnlichen triggerfreien Ansatz mit Kollisionsraten bis zu 10 MHz, was eine effiziente Softwaretriggerung und Echtzeitrekonstruktion erfordert.

In dieser Arbeit wurden drei Hauptbeiträge geleistet. Für das ALICE-Experiment wurden mehrere Komponenten der TPC-Rekonstruktionskette optimiert, darunter Zero-Suppression-Dekodierung, eine parallelisierte Track-Merger-Implementation und ein effizientes Cluster-Gathering. Diese Optimierungen ermöglichen dem O^2 Framework die Echtzeitrekonstruktion bei den erforderlichen hohen Kollisionsraten.

Als zweiter Beitrag wurde xpu entwickelt, eine schlanke C++-Bibliothek für portable GPU-Programmierung über CUDA-, HIP- und SYCL-Backends. Durch separate Kompilierung von Device-Code und dynamische Backend-Auswahl erreicht xpu nahezu native Leistung auf verschiedenen Architekturen bei vernachlässigbarem Overhead, ohne Kompromisse bei der Vorhersagbarkeit einzugehen.

Drittens wurde eine vollständige GPU-basierte Rekonstruktionskette für das Silicon Tracking System des CBM-Experiments entwickelt. Die Implementierung erreicht eine 122-fache Beschleunigung gegenüber dem bisherigen CPU-Code durch parallele Algorithmen für Cluster-Finding, Hit-Rekonstruktion und einen optimierten Merge-Sort. Bei der mCBM-Strahlzeit im Mai 2024 verarbeitete das System zuverlässig bis zu 2.4 GB/s und demonstrierte damit die Machbarkeit der Hochraten-Datenverarbeitung für den zukünftigen CBM-Betrieb.

Optimierung der TPC-Rekonstruktion in ALICE

Das ALICE Experiment

Das ALICE-Experiment (A Large Ion Collider Experiment) ist eines der vier Hauptexperimente am Large Hadron Collider (LHC) des CERN. Im Gegensatz zu den anderen LHC-Experimenten wurde ALICE speziell für die Untersuchung von Schwerionenkollisionen konzipiert. Das Hauptziel besteht darin, die Eigenschaften des Quark-Gluon-Plasmas zu erforschen, eines Materiezustands, der Mikrosekunden nach dem Urknall existierte und in dem sich Quarks und Gluonen frei bewegten, bevor sie sich zu Hadronen zusammenschlossen.

Mit dem Beginn von LHC Run 3 im Jahr 2022 wurde ALICE grundlegend aufgerüstet, um Kollisionsraten von bis zu 50 kHz bei Blei-Blei-Kollisionen zu bewältigen. Ein zentraler Bestandteil dieses Upgrades war die Einführung einer kontinuierlichen Auslese anstelle des bisherigen triggerbasierten Systems. Dies führt zu Datenraten von mehr als 1 TB/s, die in Echtzeit verarbeitet werden müssen. Zur Bewältigung dieser enormen Datenmengen wurde ein neues Rechencluster, die Event Processing Node (EPN) Farm, mit 2800 GPUs installiert.

Der TPC Detektor

Die Time Projection Chamber (TPC) ist der zentrale Spurdetektor des ALICE-Experiments. Es handelt sich um eine große zylindrische Kammer mit einer Länge von 5 m und einem Durchmesser von 5 m, die mit einem Gasgemisch gefüllt ist. Wenn geladene Teilchen die TPC durchqueren, ionisieren sie das Gas entlang ihrer Bahn und erzeugen Elektronenspuren, die unter dem Einfluss eines elektrischen Feldes zu den Endkappen driften. An den Endkappen werden diese Signale verstärkt und detektiert, wodurch die Teilchenbahnen im dreidimensionalen Raum mit hoher Präzision rekonstruiert werden können.

Die rekonstruierten Spuren aus der TPC liefern wichtige Informationen zur Teilchenidentifikation und Impulsbestimmung, was sie zu einem wesentlichen Bestandteil der ALICE-Datenanalyse macht. Die Verarbeitung der TPC-Daten stellt jedoch die größte Herausforderung für das Rekonstruktionssystem dar und beansprucht etwa 90 % der verfügbaren GPU-Rechenleistung während der Datennahme.

Zero-Suppression-Dekodierung

Die TPC-Daten werden mittels Zero-Suppression-Kodierung übertragen, um die erforderliche Bandbreite zu reduzieren. Bei diesem Format werden nur Kanäle gespeichert, die Ladungsablagerungen über einem Schwellenwert registriert haben, organisiert in 8 kB großen Seiten. Die effiziente Dekodierung dieser Daten auf GPUs stellt

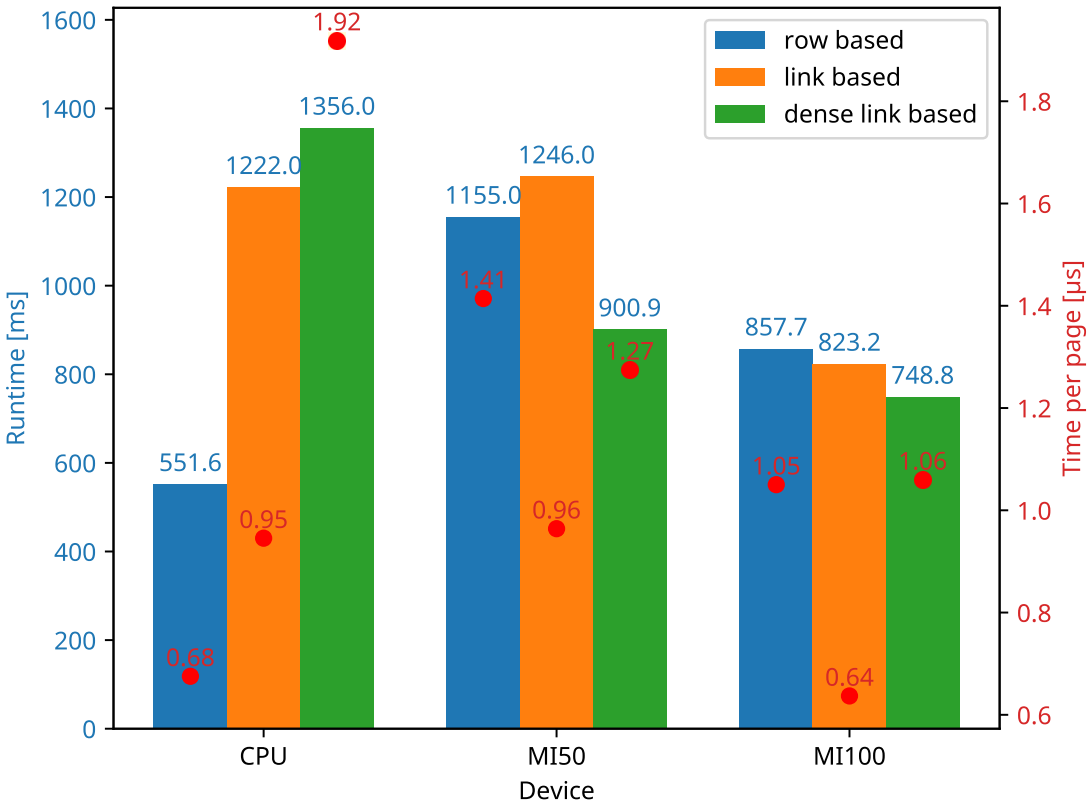

Abbildung 1: Laufzeitvergleich der verschiedenen Zero-Suppression-Formate auf der MI50-GPU, der MI100-GPU und der CPU. Die vertikalen Balken zeigen die Gesamtzeit für die Dekodierung des gesamten Timeframes, während die roten Markierungen die durchschnittliche Zeit pro Seite darstellen. Trotz höherer Verarbeitungszeit pro Seite erreicht das komprimierte Dense-Link-Format die beste Gesamtleistung auf GPUs.

mehrere Herausforderungen dar, insbesondere hinsichtlich der Speicherzugriffsmuster und der parallelen Verarbeitung.

Im Laufe der Zeit hat sich das TPC-Zero-Suppression-Format weiterentwickelt, um die steigenden Datenraten in Run 3 besser bewältigen zu können. Das ursprüngliche zeilenbasierte Format wurde 2022 durch eine linkbasierte Version ersetzt, die später zum dichteren linkbasierten Format verfeinert wurde, um bessere Kompressionsraten für Hochintensitäts-Blei-Blei-Kollisionen zu erreichen.

Die GPU-Implementierung des Dekoders nutzt eine zweistufige Parallelisierung: auf Seitenebene durch Blockzuweisung und innerhalb jeder Seite auf Kanalebene durch Thread-Zuweisung. Der Dekoder führt zunächst einen parallelen Scan über die Kanalbitmaske durch, um die Ausgabepositionen für die Daten jedes Kanals zu bestimmen. Diese Scan-Operation wird unter Verwendung von Warp-Level-Primitiven effizient implementiert, um den Synchronisationsaufwand zu minimieren.

Wie in Abbildung 1 dargestellt, zeigen die verschiedenen Formate unterschiedliche Leistungscharakteristika auf GPUs und CPUs. Das dichter kodierte Format erfordert weniger Speicherbandbreite und erzielt damit trotz höherer Verarbeitungszeit pro Seite die beste Gesamtleistung auf GPUs. Interessanterweise ist das dichter kodierte Format auf der CPU am langsamsten, aber auf GPUs am schnellsten, was die Unterschiede in den Optimierungszielen und Architekturmerkmalen zwischen CPU- und GPU-Verarbeitung verdeutlicht.

Parallelisierung des Track-Mergers

Der Track-Merger-Kernel kombiniert Spurabschnitte zu vollständigen Spuren basierend auf einer Liste von Verbindungen. In der ursprünglichen sequentiellen Implementierung war dieser Prozess ein Engpass, der die Gesamtleistung der Rekonstruktionskette beeinträchtigte. Eine direkte Parallelisierung wäre aufgrund der potenziellen Konflikte beim gleichzeitigen Schreiben in dieselben Spurabschnitte schwierig und würde komplexe Synchronisationsmechanismen erfordern.

Statt zu versuchen, den Track-Merge-Prozess selbst zu parallelisieren, wurde ein alternativer Ansatz gewählt, der unabhängige Mengen von Spurabschnitten identifiziert. Zwischen diesen Mengen sind keine Kollisionen möglich, sodass sie parallel verarbeitet werden können. Spurabschnitte und Verbindungen können als Knoten und Kanten eines Graphen interpretiert werden. Die zusammenhängenden Komponenten dieses Graphen bilden die unabhängigen Mengen, die gesucht werden.

Zur Identifizierung dieser zusammenhängenden Komponenten wurde ein effizienter GPU-Algorithmus implementiert. Jeder GPU-Block verarbeitet dann separate zusammenhängende Komponenten, um Spurabschnitte zu vollständigen Spuren zu verschmelzen. Dieser Ansatz vermeidet die Notwendigkeit von Sperren oder komplexen Konfliktlösungsstrategien.

Die Leistungsverbesserung dieser parallelen Implementierung ist bemerkenswert. Auf der AMD MI50 GPU wurde eine Beschleunigung um das 30-fache erreicht, wobei die Ausführungszeit von 6739 ms auf 204.9 ms reduziert wurde. Auf der leistungsfähigeren MI100 GPU wurde sogar eine 47-fache Beschleunigung erzielt, von 7607 ms auf 159 ms.

Cluster-Gathering-Optimierungen

Nach der Spurrekonstruktion müssen die TPC-Cluster vom GPU-Speicher zurück in den Host-Speicher kopiert werden. Die Cluster sind in einem Structure-of-Arrays (SoA)-Format gespeichert und im Speicher entsprechend ihrer Spurzuordnung verstreut. Ein naiver Ansatz zum Kopieren dieser Cluster würde Millionen von kleinen Speichertransferoperationen erfordern, was äußerst ineffizient wäre.

Es wurden mehrere Strategien implementiert, um diese Datenbewegung effizient zu gestalten. Der ursprüngliche Ansatz nutzte Direct Memory Access (DMA), um direkt in gepinnten Host-Speicher zu schreiben. Um die Auswirkungen auf die gleichzeitige Verarbeitung zu minimieren, verwendet dieser Kernel nur 1-2 Blöcke, die jeweils eine einzelne Compute-Einheit belegen, wodurch der Datentransfer im Hintergrund ausgeführt werden kann, während der nächste Timeframes verarbeitet wird.

Eine verbesserte Version führt Shared-Memory-Pufferung ein, um Speichertransaktionen zu optimieren. Der Kernel sammelt zunächst Cluster in Shared-Memory-Puffern. Sobald ein Puffer voll ist, wird er in einer einzigen Operation mit Schreibvorgängen von 128 B in den Host-Speicher geschrieben. Diese Pufferungsstrategie hilft, Speicherschreibvorgänge zusammenzufassen und reduziert die Gesamtzahl der DMA-Operationen.

Die aktuelle Implementierung, der multiBlock-Kernel, nutzt die gesamte GPU. Anstatt direkt in den Host-Speicher zu schreiben, komprimiert dieser Kernel zunächst die verstreuten Cluster-Daten in zusammenhängenden Gerätespeicher. Die Arbeit wird zwischen Cluster-Typen aufgeteilt: angehängte Cluster (die zu Spuren gehören) und nicht angehängte Cluster. Blöcke mit ungerader Nummer bearbeiten nicht angehängte Cluster, während Blöcke mit gerader Nummer angehängte Cluster verarbeiten, was eine bessere Lastverteilung ermöglicht.

Durch diese Optimierungen wurde eine dramatische Leistungssteigerung erreicht. Der naive Ansatz benötigte etwa 200 s pro Timeframe, während der optimierte multiBlock-Kernel die gleiche Aufgabe in nur etwa 220 ms erledigt - eine Beschleunigung um das 1000-fache. Diese Verbesserung ist für die Echtzeitverarbeitung der TPC-Daten in ALICE entscheidend und ermöglicht es dem Rekonstruktionssystem, mit den hohen Datenraten von Run 3 Schritt zu halten.

xpu: Eine C++-Bibliothek für portablen GPU-Code

Motivation

Moderne Hochenergiephysik-Experimente nutzen zunehmend GPU-Beschleunigung, um ihre anspruchsvollen Datenverarbeitungsanforderungen zu bewältigen. Die Entwicklung GPU-beschleunigter Software für diese Experimente stellt jedoch einzigartige Herausforderungen dar: Der Code muss portabel zwischen verschiedenen GPU-Architekturen sein, über lange Zeiträume wartbar bleiben und hohe Leistung ohne Einbußen bei der Flexibilität erzielen. Darüber hinaus muss der Code für Entwicklungs- und Testzwecke effizient auf CPUs laufen und sich gleichzeitig in bestehende C++-Codebasen integrieren lassen.

Im Rahmen dieser Dissertation wurde xpu als leichtgewichtige C++-Bibliothek entwickelt, um diese Herausforderungen zu bewältigen. Ein besonderer Fokus lag auf den Anforderungen des Online-Verarbeitungssystems des CBM-Experiments, das die Entwicklung prägte und gleichzeitig als Testumgebung für die Bibliothek diente. Sie bietet eine einheitliche Schnittstelle für CUDA, HIP und SYCL, ermöglicht die Ausführung von Code auf CPUs und erlaubt Entwicklern, portablen GPU-Code zu schreiben, ohne Kompromisse bei der Leistung oder der Kontrolle über hardwarespezifische Optimierungen einzugehen.

Architektur

Die Architektur von xpu basiert auf drei Schlüsselkomponenten: einem Laufzeitsystem, Backend-Treibern und Gerätebibliotheken. Das Laufzeitsystem verwaltet die Geräteerkennung, Speicherzuweisung und Kernel-Ausführung, während die Backend-Treiber für jede unterstützte Plattform (CUDA, HIP, SYCL, CPU) als separate Shared Library kompiliert werden. Diese Treiber implementieren die vom Laufzeitsystem definierte gemeinsame Schnittstelle und ermöglichen die dynamische Auswahl des Backends zur Laufzeit.

Der Gerätecode in xpu wird in separaten Bibliotheken organisiert, die für jedes Backend unabhängig kompiliert werden können. Jede Gerätebibliothek durchläuft eine duale Kompilierung: einmal als regulärer C++-Code für den Host, um Typenprüfung und Symbolauflösung zu ermöglichen, und ein weiteres Mal als Gerätecode für jedes verfügbare Backend mit dem jeweiligen Compiler. Dieser Ansatz ermöglicht es xpu, den optimalen Compiler für jede Plattform zu verwenden (z.B. nvcc für CUDA und hipcc für HIP), während regulärer C++-Code für die CPU-Ausführung ohne zusätzliche Abhängigkeiten beibehalten wird.

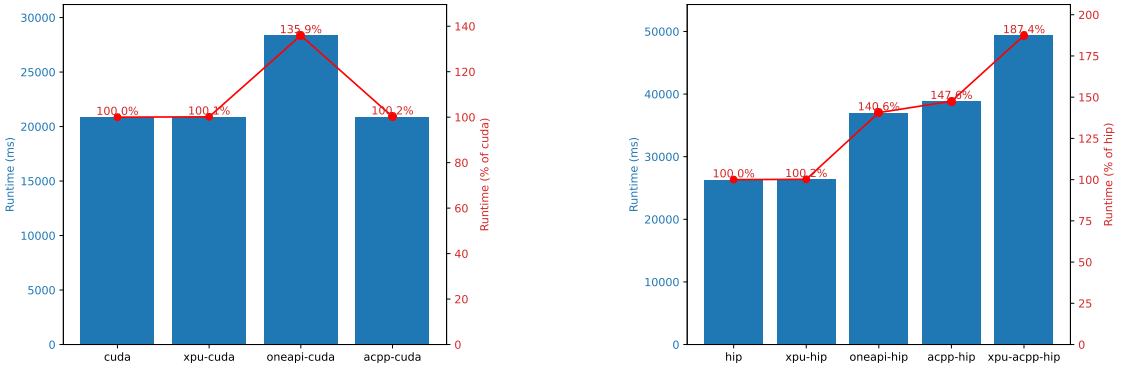

Abbildung 2: Laufzeit vergleich der verschiedenen Implementierungen des STS Unpackers.

Kernfunktionalität

Die Kernfunktionalität von xpu umfasst RAII-basierte Speicherverwaltung, die explizite Kontrolle über Allokationen ermöglicht. Im Gegensatz zu SYCL, das Speicherverwaltung zu abstrahieren versucht und damit die Kontrolle über Speicheroperationen einschränkt, bietet xpu Buffer-Klassen, die wie rohe Zeiger im Gerätecode verwendet werden können, ohne Overhead einzuführen. Verschiedene Speichertypen (Gerät, Host-gepinnt, verwaltet, Ein-/Ausgabe) werden unterstützt, um unterschiedliche Zugriffsmuster zu berücksichtigen.

Kernel werden als aufrufbare Objekte implementiert, die von der Klasse xpu::kernel erben. Jeder Kernel erhält ein Context-Objekt, das Informationen über die Position des Threads im Ausführungsgitter, Zugriff auf Shared Memory und Zugriff auf Constant Memory bietet. Die Parallelitätsprimitiven auf Block- und Warp-Ebene wie Reduktionen, Scans und Ballot-Operationen werden über alle Backends hinweg konsistent unterstützt.

Für die Integration nativer GPU-Funktionalität stellt xpu einen Mechanismus für Host-Funktionen bereit. Dies ermöglicht die Verwendung plattformspezifischer Implementierungen, die separat für jedes Backend kompiliert werden, während die Typsicherheit gewährleistet bleibt.

Performanz

Um den potenziellen Leistungsoverhead von xpu und SYCL zu untersuchen, wurde der GPU-Port des STS-Unpackers als Benchmark verwendet. Der Unpacker ist der erste Schritt in der Verarbeitungskette und dekontextualisiert den Strom der Rohdaten des Detektors, was für das nachfolgende Cluster-Finding erforderlich ist. Der Benchmark wurde auf einer NVIDIA RTX 2080 Ti und einer AMD MI50 durchgeführt, wobei 30 Timeslices aus der mCBM-Kampagne 2022 als Testdaten verwendet wurden.

Wie in Abbildung 2 dargestellt, erreicht die xpu-Version im Wesentlichen die Leistung beider nativen Implementierungen, mit nur einem Overhead von 0.2 % in der Laufzeit im Vergleich zu den jeweiligen CUDA- und HIP-Varianten. Der generierte Assembler-Code ist nahezu identisch; nur die Reihenfolge der Anweisungen unterscheidet sich leicht.

Die Leistungsunterschiede zwischen SYCL-Implementierungen lassen sich auf ihre unterschiedlichen Compiler-Architekturen zurückführen. Intel oneAPI und AdaptiveCpp bauen beide auf LLVM/Clang auf, verfolgen aber grundlegend unterschiedliche Ansätze. Intel oneAPI verwendet einen stark angepassten Clang-Fork mit proprietären Optimierungen, was seine konsistentere, aber langsamere Leistung über Plattformen hinweg erklärt - etwa 36 % langsamer auf CUDA mit einer ähnlichen Leistungslücke auf ROCm. AdaptiveCpp, kompiliert mit Clang 18, verwendet eine Plugin-Architektur zur Generierung von Backend-Code. Während dieser Ansatz eine bessere Wartbarkeit und einfachere Updates auf neue Clang-Versionen verspricht, deuten die Leistungsergebnisse darauf hin, dass die Übersetzungsschicht von SYCL zu nativem GPU-Code erheblichen Overhead einführt, insbesondere für AMD-Hardware, wo die Implementierung 47 % langsamer als natives HIP läuft.

GPU-beschleunigte STS-Rekonstruktion für CBM

Das CBM-Experiment

Das Compressed Baryonic Matter (CBM) Experiment, derzeit im Bau an der FAIR-Anlage bei GSI, zielt darauf ab, die Eigenschaften stark wechselwirkender Materie unter extremen Bedingungen von Temperatur und Dichte durch Schwerionenkollisionen zu erforschen. Die Silicon Tracking System (STS) Rekonstruktion in CBM steht vor erheblichen rechnerischen Herausforderungen aufgrund des frei strömenden Datenerfassungsansatzes. Im Gegensatz zu herkömmlichen getriggerten Systemen muss CBM den kompletten Rohdatenstrom in Echtzeit bei Wechselwirkungsraten von bis zu 10 MHz verarbeiten. Als primärer Spurendetektor spielt das STS eine entscheidende Rolle bei der Spurrekonstruktion, wodurch seine effiziente Online-Verarbeitung für die Echtzeit-Ereignisauswahl von kritischer Bedeutung ist.

Der STS-Detektor

Das Silicon Tracking System (STS) ist eine zentrale Komponente des CBM-Experiments. Positioniert innerhalb des Magneten und stromaufwärts des Targets ist es der erste Detektor nach dem Beam Monitor zusammen mit dem MVD. Das STS besteht aus doppelseitigen Silizium-Streifensensoren, die auf 8 Ebenen (auch als Stationen bezeichnet) verteilt sind und die Spurrekonstruktion und Impulsbestimmung geladener Teilchen ermöglichen. Die hohe zeitliche und räumliche Auflösung von 5 ns und 25 µm

machen es zu einem Schlüsseldetektor für die Spurrekonstruktion und damit besonders wichtig für die Online-Verarbeitung.

Die STS Rekonstruktion

Der STS-Hitfinder in seiner sequentiellen Form transformiert Rohdaten des Detektors in vierdimensionale Raum-Zeit-Punkte für die Spurrekonstruktion. Der Cluster-Finding-Algorithmus verarbeitet nach Zeit sortierte Digis, indem er jede Messung sequentiell untersucht. Bei einem neuen Digi wird geprüft, ob bereits ein Cluster auf benachbarten Streifen innerhalb eines konfigurierbaren Zeitfensters gebildet wird. Ist dies der Fall, wird das Digi zu diesem Cluster hinzugefügt; andernfalls wird ein neuer Cluster begonnen. Die Implementierung teilt diesen Prozess in zwei Schritte: Zunächst werden die zu einem Cluster gehörenden Digis identifiziert und deren Indizes gespeichert, bevor in einem zweiten Durchlauf die Cluster-Eigenschaften berechnet werden.

Der Hit-Finding-Algorithmus kombiniert Cluster von beiden Sensorseiten zu dreidimensionalen Positionen. Für jede potenzielle Kombination von Front- und Back-Side-Clustern prüft der Algorithmus zunächst, ob ihre Zeitmessungen innerhalb eines konfigurierbaren Fensters kompatibel sind. Bei zeitlich passenden Clustern berechnet der Algorithmus die geometrischen Schnittpunkte entlang der überlappenden Streifen. Die zeitliche Übereinstimmung reduziert signifikant die Anzahl der Cluster-Kombinationen, die geometrisch ausgewertet werden müssen, was die Recheneffizienz verbessert. Für jeden rekonstruierten Hit muss der Algorithmus auch die Positionsunsicherheiten von den Eingangs-Clustern auf die endgültigen Hit-Koordinaten übertragen, wobei sowohl die Unsicherheiten in den Cluster-Positionen als auch die geometrischen Effekte des Stereo-Winkels berücksichtigt werden.

Parallele STS Rekonstruktion

Die parallele STS-Rekonstruktion verwendet mehrere grundlegende Optimierungen, die sich auf effiziente Datenstrukturen und Speicherverwaltung konzentrieren. Diese Optimierungen sind besonders wichtig, da die Rekonstruktion zwischen 10^8 und 10^9 Hits pro Timeslice in Echtzeit während der Datenaufnahme verarbeiten muss.

Eine primäre Optimierung umfasst die vollständige Neugestaltung der für die Rekonstruktion verwendeten Datenstrukturen. Die Offline-Rekonstruktion verwendet eine hierarchische Klassenstruktur, die tief in ROOT integriert ist. Die Online-Rekonstruktion implementiert stattdessen leichtgewichtige C++-Strukturen, die diese Abhängigkeiten eliminieren. Durch die Entfernung der ROOT-Integration und die Abflachung der Vererbungshierarchie wird der Speicherbedarf der grundlegenden Datentypen erheblich reduziert. So wurde beispielsweise die Größe eines Cluster-Objekts von 112 Byte

in der Offline-Rekonstruktion auf nur 24 Byte in der Online-Version reduziert. Ähnlich wurden Hit-Objekte von 136 Byte auf 48 Byte reduziert.

Eine weitere wesentliche Optimierung betrifft die Behandlung von Digi-Indizes während der Cluster-Erstellung. Die Offline-Rekonstruktion reserviert zusätzlichen Speicher, um Indizes von beitragenden Digis für jeden Cluster zu speichern, was sowohl den Speicher für ein `std::vector`-Objekt (24 Byte) als auch eine zusätzliche Heap-Allokation erfordert. Die Online-Implementierung eliminiert diesen Overhead, indem sie Cluster-Eigenschaften direkt während der Konstruktion berechnet.

Die Rekonstruktionskette nutzt auch Möglichkeiten für eine erhöhte Parallelisierung. Während die Offline-Version typischerweise Daten auf Modulebene verarbeitet, kann die Online-Implementierung stattdessen über Modulseiten parallelisieren, wo dies angemessen ist, was die verfügbare Parallelität für diese Operationen effektiv verdoppelt.

Der parallele Cluster-Finding-Algorithmus implementiert eine zweiphasige Strategie, um eine effiziente parallele Verarbeitung auf GPUs zu ermöglichen. Anstatt zu versuchen, vollständige Cluster in einem einzigen Schritt zu identifizieren, was eine komplexe Synchronisierung zwischen Threads erfordern würde, etabliert der Algorithmus zunächst Verbindungen zwischen Digis, die zum selben Cluster gehören, und erstellt dann die eigentlichen Cluster-Objekte in einer zweiten Phase.

Die primäre Änderung am Hit-Finding-Algorithmus bestand darin, die parallele Arbeit im Vergleich zur Offline-Implementierung zu erhöhen. Während die Offline-Version Daten auf Modulebene mit einem CPU-Thread pro Modul verarbeitet, weist die GPU-Implementierung jedem Front-Side-Cluster einen Thread zu, was eine vollständige Parallelisierung über alle potenziellen Hits ermöglicht.

Performanz

Die Gesamtleistungsmerkmale der Hitfinder-Implementierungen sind in Abbildung 3 dargestellt. Bei der Offline-Implementierung unterscheidet die Abbildung zwischen der parallelen Rekonstruktionsphase (hellrot) und den sequentiellen Overhead-Phasen (dunkelrot). Diese sequentiellen Phasen umfassen die anfängliche Verteilung von Digis auf Module und die abschließende Sammlung von Clustern und Hits in zusammenhängende Ausgabearrays.

Bei der Betrachtung der Single-Thread-Leistung benötigt die Offline-Implementierung 11.67 s Gesamtlaufzeit, wobei 9.4 s im parallelen Abschnitt und 2.3 s im sequentiellen Overhead verbracht werden. Die Online-Implementierung erreicht mit 5.48 s eine bessere Basisleistung, was einer 2,1-fachen Beschleunigung entspricht, bevor irgendeine Parallelisierung angewendet wird.

Beide Implementierungen zeigen eine effektive Skalierung mit zunehmender Thread-Anzahl. Der parallele Abschnitt der Offline-Version skaliert weiterhin bis zu 64 Threads,

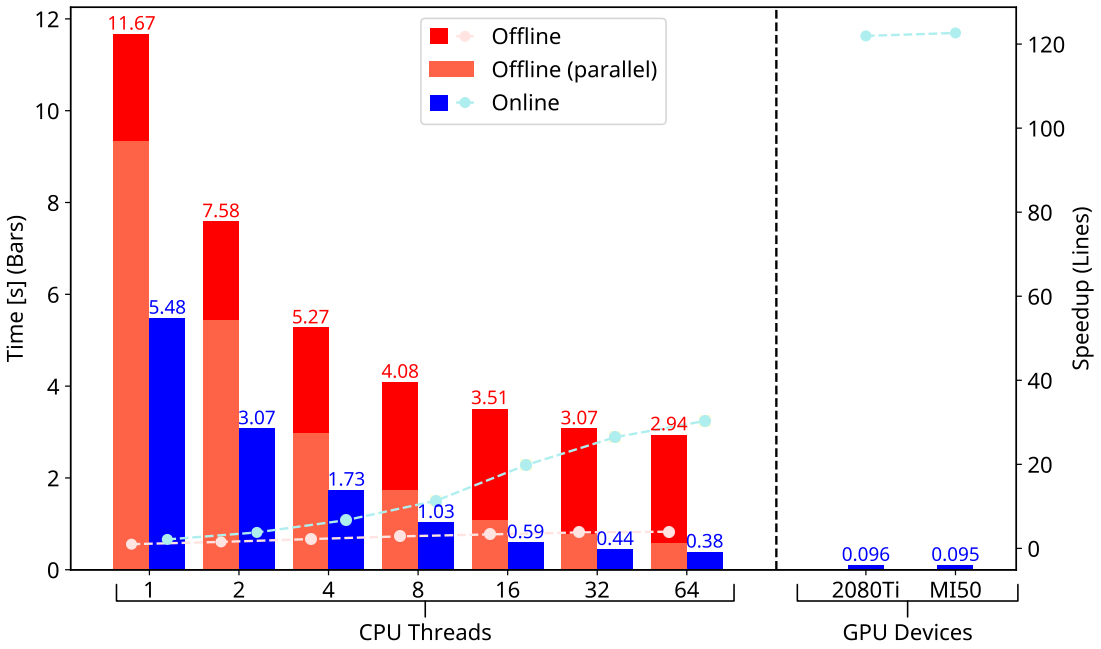

Abbildung 3: Laufzeitvergleich der offline und online Varianten des STS Hitfinders.

was zeigt, dass der Rekonstruktionsalgorithmus selbst gut parallelisierbar ist. Die Gesamtleistung wird jedoch zunehmend vom sequentiellen Overhead dominiert. Dies schafft einen Leistungsengpass, der die maximal erreichbare Beschleunigung für die Offline-Implementierung begrenzt.

Die Online-Implementierung löst diese Einschränkung, indem sie sowohl die Rekonstruktion als auch die Datenbewegungsphasen parallelisiert. Dieser Ansatz behält die Skalierung über den gesamten Bereich der getesteten Thread-Anzahlen bei und erreicht 0.38 s bei 64 Threads, was einer 14,4-fachen Beschleunigung gegenüber der sequentiellen Baseline entspricht.

Die GPU-Ausführung bietet eine weitere erhebliche Beschleunigung. Beide getesteten Geräte, die NVIDIA RTX 2080 Ti und AMD MI50, erreichen eine ähnliche Leistung von etwa 0.095 s. Dies entspricht ungefähr einer 122-fachen Beschleunigung im Vergleich zur sequentiellen Offline-Ausführung und ist 4,0-mal schneller als die beste CPU-Leistung der Online-Version. Die konsistente Leistung über diese verschiedenen GPU-Architekturen hinweg demonstriert die Fähigkeit des Algorithmus, verschiedene Hardware-Plattformen effektiv zu nutzen.

Einsatz in Produktionsumgebung

Die Entwicklungen der Online-Software gipfelten im ersten Produktionseinsatz während der mCBM-Strahlzeit im Mai 2024. Vier Tage lang lief die Online-Rekonstruktion kontinuierlich parallel zur Aufzeichnung von Timeslices mit Detektorrohdaten auf Festplatten. Dies stellte sicher, dass unverarbeitete Daten noch für spätere Offline-Analysen verfügbar waren, während es gleichzeitig die erste Demonstration der Online-Verarbeitungsfähigkeiten in CBM darstellte.

Das System demonstrierte robuste Leistung bei der Verarbeitung kontinuierlicher Datenströme. Die durchschnittliche Datenrate erreichte etwa 800 MB/s, mit Spitzen bis zu 2.4 GB/s. Der STS-Detektor trug mit einem Durchschnitt von 310 MB/s den größten Anteil bei und erreichte Spitzen von 900 MB/s.

Die erfolgreiche Einführung während der mCBM-Strahlzeit im Mai 2024 demonstrierte, dass die Rekonstruktionskette effektiv unter realen Bedingungen arbeiten konnte, wobei Daten von tatsächlichen Detektoren anstelle von idealisierten Simulationen verarbeitet wurden. Das System behielt trotz variierender Strahlbedingungen, einschließlich schwankender Datenraten entsprechend dem SIS18-Extraktionszyklus, Detektorrauschen und sich entwickelnder Kalibrierungsparameter, eine stabile Leistung bei. Diese betriebliche Validierung lieferte entscheidende Beweise dafür, dass die Rekonstruktionsalgorithmen robust genug sind, um die unvorhersehbaren Aspekte der experimentellen Datenerfassung zu bewältigen, die in Simulationen nicht vollständig erfasst werden können. Der kontinuierliche Betrieb über vier Tage bestätigte weiter die Zuverlässigkeit des Systems für längere Produktionsläufe und schuf eine solide Grundlage für die zukünftige Online-Verarbeitungsinfrastruktur des CBM-Experiments.

Fazit

Diese Dissertation präsentiert mehrere wesentliche Beiträge zur GPU-Beschleunigung von Hochenergiephysik-Experimenten. Für ALICE wurden verschiedene Komponenten der TPC-Rekonstruktionskette optimiert, was eine Echtzeitverarbeitung bei Kollisionsraten von 50 kHz ermöglicht. Die entwickelte xpu-Bibliothek bietet eine portable Lösung für GPU-Programmierung mit vernachlässigbarem Overhead und unterstützt verschiedene Hardware-Architekturen durch separate Kompilierung von Device-Code. Für das CBM-Experiment wurde eine vollständige GPU-beschleunigte Rekonstruktionskette implementiert, die eine 122-fache Beschleunigung gegenüber dem CPU-basierten Code erreicht und erfolgreich während der mCBM-Strahlzeit im Mai 2024 eingesetzt wurde.

Diese Entwicklungen finden in einem Umfeld statt, in dem sich die Hardware-Fähigkeiten kontinuierlich weiterentwickeln. Moderne CPU-Architekturen mit AVX-512-Vektoranweisungen könnten für bestimmte Workloads eine Alternative zur GPU-Verar-

beitung darstellen. Die Ergebnisse in beiden Experimenten zeigen, dass eine sorgfältige algorithmische Optimierung in Verbindung mit GPU-Beschleunigung Leistungsverbesserungen um Größenordnungen erzielen kann. Die in dieser Arbeit demonstrierten Echtzeit-Verarbeitungsfähigkeiten etablieren eine Grundlage, um den zukünftigen Rechenanforderungen von Hochenergiephysik-Experimenten gerecht zu werden.

GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

Publiziert unter der Creative Commons-Lizenz Namensnennung (CC BY) 4.0 International.
Published under a Creative Commons Attribution (CC BY) 4.0 International License.
https://creativecommons.org/licenses/by/4.0/