Coprocessor integration for real-time event processing in particle physics detectors

Alexey Pavlovich Badalov

http://hdl.handle.net/10803/396128

TESIS DOCTORAL

Título	Coprocessor integration for real-time event processing in particle physics detectors
Realizada por	Alexey Badalov
en el Centro	La Salle – Ramon Llull University
y en el Departamento	GR-SETAD
Dirigida por	Dr. Xavier Vilasis i Cardona Dr. Niko Neufeld

Coprocessor integration for real-time event processing in particle physics detectors

Alexey Badalov

Abstract

High-energy physics experiments today have higher energies, more accurate sensors, and more flexible means of data collection than ever before. Their rapid progress requires ever more computational power; and massively parallel hardware, such as graphics cards, holds the promise to provide this power at a much lower cost than traditional CPUs. Yet, using this hardware requires new algorithms and new approaches to organizing data that can be difficult to integrate with existing software.

In this work, I explore the problem of using parallel algorithms within existing CPU-orientated frameworks and propose a compromise between the different trade-offs. The solution is a service that communicates with multiple event-processing pipelines, gathers data into batches, and submits them to hardware-accelerated parallel algorithms.

I integrate this service with Gaudi — a framework underlying the software environments of two of the four major experiments at the Large Hadron Collider. I examine the overhead the service adds to parallel algorithms. I perform a case study of using the service to run a parallel track reconstruction algorithm for the LHCb experiment's prospective VELO Pixel subdetector and look at the performance characteristics of using different data batch sizes. Finally, I put the findings into perspective within the context of the LHCb trigger's requirements.

Table of Contents

1 Overview 6
2 Large Hadron Collider beauty experiment 9
2.1 Large Hadron Collider 9
2.2 The LHCb experiment 10
2.3 LHCb Computing 14
3 LHCb software infrastructure 18
3.1 Requirements 18
3.2 Gaudi 19
3.3 Software organization 21
3.4 Event data model 21
3.5 Gaudi Hive 22
4 The need for speed 26
4.1 High-luminosity upgrade 26
4.2 Computation alternatives 27
4.3 Intel Xeon Phi 27
4.4 GPU programming 28
5 State of the art 31
5.1 NA62 31
5.2 NA62 NaNet 32
5.3 ALICE 35
5.4 ATLAS 37
5.5 ATLAS APE 39
6 Towards a new system 41
6.1 Coprocessor access 41
6.2 Interprocess communication 42
6.3 Data management 44
6.4 Gaudi integration 45
6.5 Apache Thrift experiment 45
7 Coprocessor Manager 47
7.1 Overview 47
7.2 Design 48
7.3 Gaudi Hive considerations 52
7.4 Components 52
8 VELO Tracking on the GPU 58

8.1 Track types in LHCb 58
8.2 Track reconstruction 59
8.3 Tracking efficiency 61
8.4 CPU-based VELO Pixel tracking 61
8.5 GPU-based VELO Pixel tracking 62
9 Performance evaluation 65
9.1 Hardware setup 65
9.2 Overhead measurement 65
9.3 Event batching 67
10 Analysis 71
10.1 Performance 71
10.2 Cost 73
11 Conclusion 75
Bibliography 77

1 Overview

We want to understand the world. For millennia, since times before Socrates, Western science was guided by various versions of the theory of elements; it was thought that all materials were made of *fire*, *earth*, *air*, and *water*, each possessing intrinsic qualities, like "coldness" and "bluntness," that they endowed the composite of which they were part. In the Middle Ages, alchemists began to discover new phenomena the existing elements could not explain and added sulfur and arsenic to the list. Eventually there were dozens of elements, and among them patterns emerged. These patterns suggested that a finer structure existed. Eventually, this structure was revealed with the discovery of subatomic particles.

This cycle of establishment of a theory, its expansion to fit new observations, and then discovery of a more fundamental layer would repeat once more to get us to today's state of the art in physics — the Standard Model of particle physics. The Standard Model describes the entities that give rise to all the matter, energy, and forces in the Universe.

We know the Standard Model is still incomplete: for one, it does not explain gravity. To advance the theory we need to keep discovering new phenomena, physics beyond the Standard Model, and this is what high-energy particle colliders like the LHC (**l**arge **h**adron **c**ollider) are built to achieve.

LHC accelerates particles along circular beams moving in opposite directions. There are four detectors placed around this circle, each housing an interaction region where particles collide with each other. Since it first started operation in 2008, LHC has gone through a series of upgrades that increased its energy and collision rates. The next major upgrade is expected to start in 2017.

A particle detector is like a video camera that takes millions of frames per second, it outputs data at very high rates. In order to determine which particle collisions are interesting and what new particles are produced in the collisions, this output data is analyzed by computers. The more collisions occur and the greater the output data rate, the more computing power is needed.

The LHCb (**L**arge **H**adron **C**ollider **b**eauty) experiment is one of the four major experiments at the LHC. Its goal is to search for new particles or forces beyond the Standard Model. The LHCb detector and computing infrastructure are being upgraded along with LHC. There is a fixed budget for the upgrade, as well as other limitations, so much work goes into designing better algorithms, making more efficient use of existing hardware, and evaluating new types of hardware for possible efficiency gains.

When evaluating new hardware, we look for it to be cost-effective relative to throughput, energy-efficient, and maintainable. It has to be significantly better than conventional solutions in order to justify the added complexity. Some candidates under investigation are Intel Xeon Phi [1] and GPUs (**g**raphics **p**rocessing **u**nits). Both of these types of coprocessors are specialized for providing high throughput per watt on large datasets and highly parallel algorithms. As of November 2015, GPUs were used in nine out of the ten top supercomputers in the Green500 list [2], which ranks the top 500 supercomputers in the world by energy efficiency.

There is a catch. It is not enough to simply install these coprocessors in the LHCb computer farm to reap the benefits. The algorithms currently in use and the software infrastructure supporting them have been designed for efficient serial execution, but not for massive parallelism. New, parallel, algorithms are required to use the new hardware, and the infrastructure needs to be updated to be capable of supplying them with appropriately structured data.

Parallel algorithm design for detector data processing is already an active research area. The NA62 experiment, ALICE, and ATLAS have written track reconstruction algorithms optimized for GPUs [3] [4] [5]. ALICE also investigated integration of GPU algorithms with the rest of their software. ATLAS, which shares much of its software infrastructure with LHCb, made several efforts into making it fully compatible with massively parallel coprocessors [5] [6] [7].

At the same time as new algorithms are being developed, LHCb's infrastructure needs to be extended in order to test the new code and ultimately put it into production, should it be deemed sufficiently beneficial. *To be useful, the combination of the new infrastructure together with the coprocessor hardware and parallel algorithms has to provide a large benefit in terms of throughput relative to the cost and it has to satisfy LHCb's performance requirements.*

Efficiency of an algorithm can be estimated by running it independently, decoupled from all the rest. However, it is also necessary to see how coprocessor algorithms run together with the existing sequential software. It is important to have the infrastructure for integrating coprocessor algorithms in place in order to validate their use.

We describe our coprocessor algorithm system, called the Coprocessor Manager. It allows using algorithms targeting new coprocessors in combination with existing proven software. It creates an environment for adding new algorithms to the existing computation pipelines and manages the flow of data in a way that lets the algorithms use parallel coprocessor hardware effectively. The extension's design is informed by lessons learned from the other experiments' efforts and makes use of a distinct architecture.

This thesis is based in part on the papers *LHCb GPU Acceleration Project* [8] and *A GPU offloading mechanism for LHCb* [9].

The rest of the document is organized into 11 chapters:

- *Chapter 2* is a guided tour of the background information, starting at CERN's Large Hadron Collider's goals and ending with the internals of LHCb experiment's computing system.
- *Chapter 3* adds details about LHCb's software infrastructure.
- *Chapter 4* describes the high-luminosity upgrade at LHC and the challenges to LHCb's computing system that it presents.
- *Chapter 5* describes previous works.
- *Chapter 6* describes the design decisions involved in designing a solution for using coprocessor algorithms.
- *Chapter 7* presents the Coprocessor Manager in full detail.
- *Chapter 8* describes a GPU-based track reconstruction algorithm developed for our extension.
- *Chapter 9* evaluates the Coprocessor Manager's performance using the track reconstruction algorithm as a benchmark.
- *Chapter 10* further analyzes the results.
- *Chapter 11* reflects in conclusion.

2 Large Hadron Collider beauty experiment

This chapter sets up the context by introducing CERN's largest particle accelerator. It explains the accelerator's purpose and gives the theoretical and historical background to introduce the physics involved. It introduces the LHCb experiment, which is the setting for this work. Finally, the chapter goes a step deeper and describes the hardware that produces the data analysis of which will be the main challenge we hope to help overcome.

2.1 Large Hadron Collider

We ourselves and all the matter we experience in everyday life are made of atoms. An atom is a nucleus, composed of protons and neutrons bound together, and surrounded by electrons. Hydrogen is made up of one proton and one electron; helium is made up of two protons, two neutrons, and two electrons; heavier element atoms contain still larger numbers of these particles. Everything that is not made from these particles consists mostly of theoretically necessary but still elusive dark matter and dark energy. Very accurate measurements of the amount of baryonic matter in the universe show that it makes up less than 4.6% of the total mass [10]. However, according to the current theory, the Standard Model of particle physics, even this is far too much — nearly all of these 4.6% should have been evenly split into matter and antimatter and mutually annihilated shortly after the Big Bang.

The very existence of matter requires that matter and antimatter behave differently from each other. This difference, called CP (**c**harge **p**arity) violation, can be quantified by dividing the observed amount of matter in the universe by the number of photons left over from matter-antimatter annihilation. This number has been found to be about one in a billion. This disagrees with the Standard Model of particle physics, which allows at most ten billion times less CP violation than we see. The model's prediction is vastly different from observations, therefore the model must be incomplete. The problem is that all of the Standard Model's predictions of events at scales and energies we can experimentally probe have proven to be highly accurate. Conditions under which the Standard Model breaks down must be more extreme.

LHC (**l**arge **h**adron **c**ollider) [11] is a particle accelerator operated by CERN — the European Organization for Nuclear Research. It is a facility used for studying high-energy particle physics. LHC is the latest in a series of ever more powerful colliders and is housed in the 26.7 km circular tunnel left over from LEP (the **l**arge **e**lectron-**p**ositron collider), its predecessor, located over 100 meters underground at the Franco-Swiss border near Geneva.

As of 2015, LHC accelerates bunches of protons (hydrogen-1 ions) or lead ions to the combined energy of 13 teraelectronvolts (TeV), moving at 99.9999997% of the speed of light (just 2.8 km/h less)[1], and smashes them together, recreating conditions similar to those soon after the Big Bang. Four major detectors sit at locations around the circular tunnel to observe particle collisions; these are: ALICE, ATLAS, CMS, and LHCb. ATLAS and CMS are general-purpose detectors, ALICE studies heavy ion collisions, and LHCb focuses on rare heavy flavoured particle interactions.

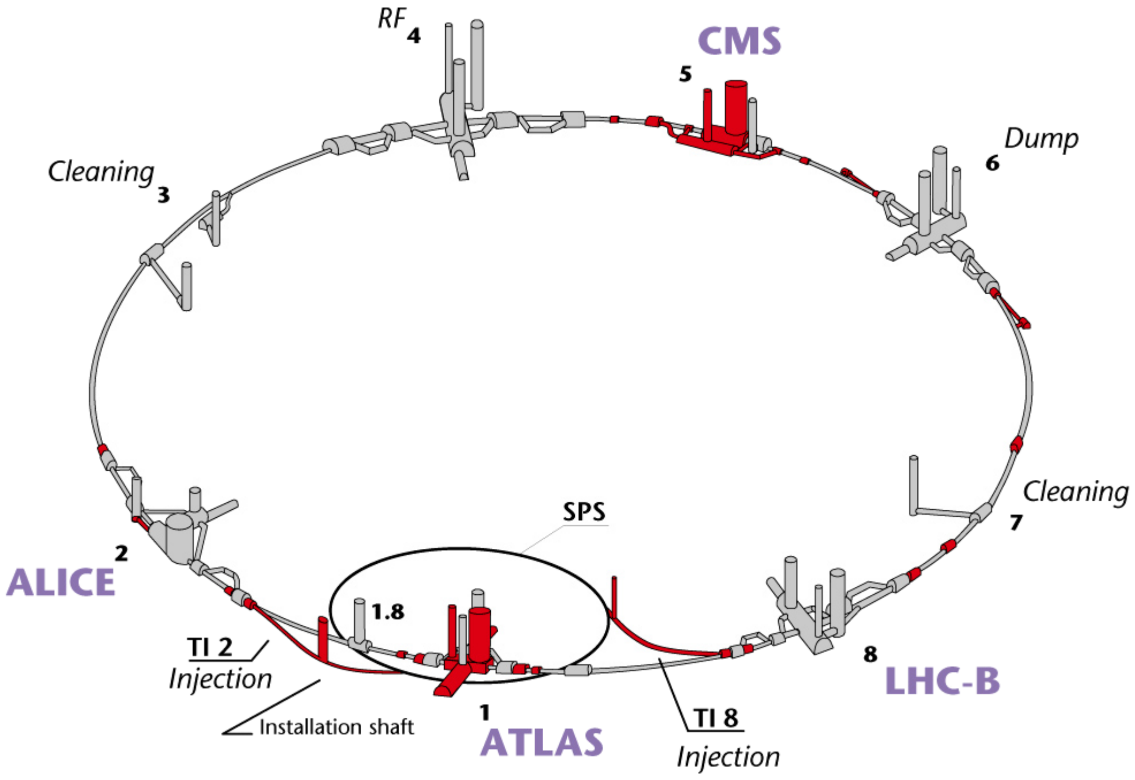

Figure 1 The LEP tunnel with LHC structures highlighted in red.

2.2 The LHCb experiment

Our understanding of the internal structure of the atom developed mainly at the turn of the 19th century. The electron was discovered in 1897 by Sir Joseph John Thomson [12] when he studied the rays emitted by electrodes inside vacuum tubes. In 1911, Ernest Rutherford used alpha radiation to probe the structure of gold atoms and proposed a model of the atom in which a small, massive nucleus was surrounded by electrons [13]. Continuing this line of investigation, Rutherford discovered the proton as part of the nucleus in 1919 [14] and hypothesized that the rest of the nucleus consisted of neutrons, later discovered by James Chadwick in 1932 [15]. These subatomic particles — proton, neutron, and electron — were thought to be indivisible, elementary.

[1] Proton speed by energy-mass equivalence at 13 TeV: $\sqrt{1-\left(1+13\text{TeV}/m_p c^2\right)^{-2}}$

This belief could not last long. During the 1950s, particle colliders produced dozens of new subatomic particles: antielectron, antiproton, muons, pions, kaons, baryons, neutrinos, and antineutrinos. In early 1960s, Murray Gell-Mann noted patterns in the properties of some of these new particles and began organizing them into a kind of a table, similar to the periodic table of chemical elements. In 1964, he showed that these patterns could be explained if the particles were composed of combinations of three smaller parts, which he called "quarks" [16]. In his paper, Gell-Mann assigned the three quarks letters u, d, and s; they would later become known as "up," "down," and "strange" quarks. Not all particles were composed of quarks, but those that were are called "hadrons". Protons and neutrons were said to be hadrons composed of up and down quarks whereas electrons remained elementary. No particle with the properties of a quark has been observed on its own, so it was proposed that quarks only existed in combinations.

In 1970, a theory called the "GIM mechanism" was proposed to explain new experimental data [17]. This theory added a fourth quark, which would later become known as "charm" on the account of having brought symmetry to the subatomic world [18] — previously, up and down quarks formed a doublet, now strange was also paired up with charm.

CP violation, introduced in section 2.1, was discovered around the same time as quarks. In 1973, Makoto Kobayashi and Toshihide Maskawa concluded that a four-quark theory of elementary particles could not explain CP violation without major revisions, but showed that a six-quark theory had potential sources of CP violation [19]. These two new quarks were initially called "truth" and "beauty" (it was romantic to set off on a search for naked truth and beauty), but the more prosaic names "top" and "bottom" would become the ones that stuck. The bottom quark was discovered in 1977 in Fermilab [20], and the top quark in 1995 [21] [22] at the same laboratory.

The LHCb (**l**arge **h**adron **c**ollider **b**eauty) [23] [24] experiment is located at Point 8 of the LHC tunnel, as illustrated in Figure 1. Its primary purpose is to search for particle behaviours beyond the Standard Model. It is specifically designed to study composite particles that contain bottom (beauty) quarks, which gives it higher precision and distinguishes it from the general-purpose ATLAS and CMS detectors.

Since LHC occupies the tunnel formerly used by the LEP collider, the existing detectors use facilities originally built for LEP. LHCb is built inside the cavern that used to house DELPHI (**de**tector with **l**epton, **p**hoton, and **h**adron **i**dentification). LHCb features the array of sensors used for track reconstruction and particle identification shown in the diagram in Figure 2.

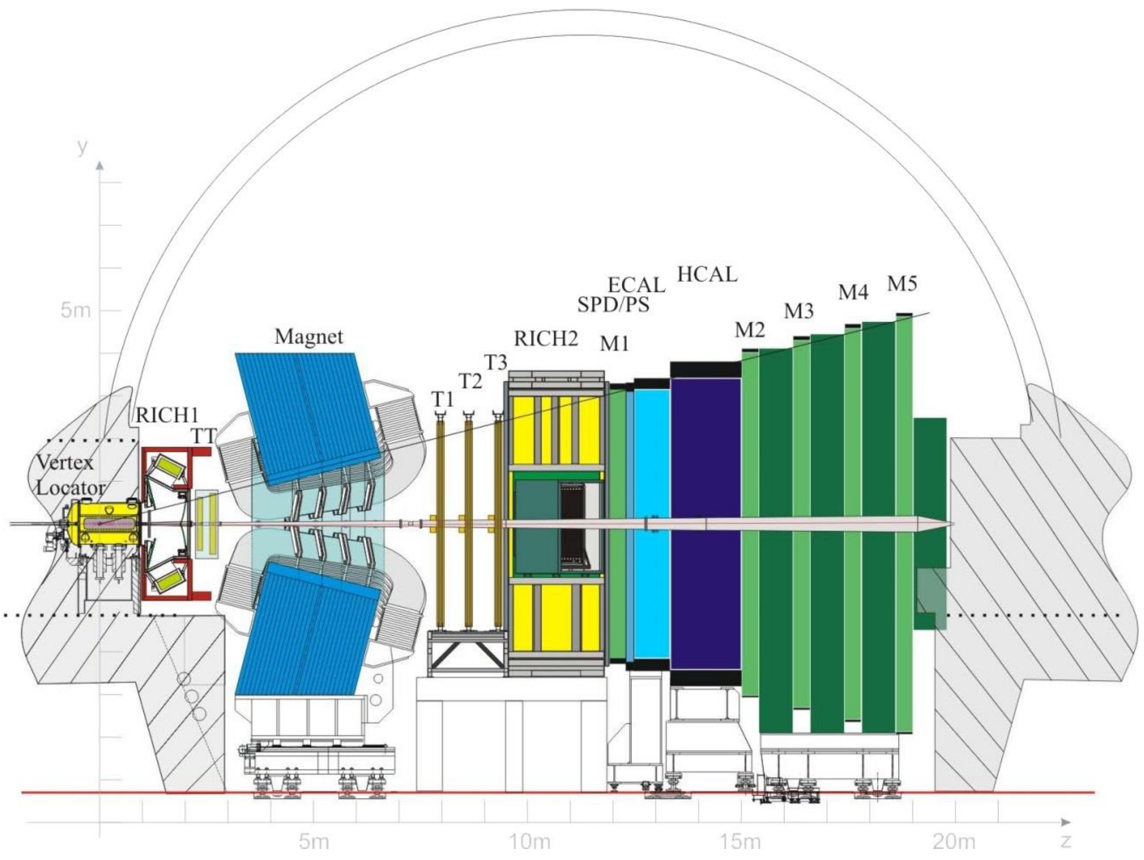

Figure 2 LHCb detector components**.** The sensors are lined up in a cone shape around the beam.

The detector is built around a particle collision site. Particles produced at the collision site initially follow straight paths, passing through the VELO (**ve**rtex **lo**cator) detector. Charged particles are then deflected by a powerful 4 Tm dipole magnet [25] to determine their momentum. The system uses RICH (**r**ing-**i**maging **Ch**erenkov detectors), downstream and upstream tracking stations, SPD (**s**cintillating **p**ad **d**etector), PS (**pres**hower), ECAL (**e**lectromagnetic **cal**orimeters), HCAL (**h**adronic **cal**orimeters), and muon detectors. Together, the moment and type completely describe each individual particle and therefore the full particle collision event.

The VELO detector is a silicon strip detector that is the part of the LHCb detector closest to the interaction region [26]. Together with the TT station, it supplies spatial track coordinates before they pass through the dipole magnet. It is also the only sensor at LHCb that measures particles produced on the side of the interaction region opposite of the rest of the detector. Its job is to measure particle tracks and precisely separate primary and secondary vertices due to decay of heavy flavoured particles. Efficient and accurate track reconstruction in the VELO is critical for vertex identification, underpinning physics analysis at LHCb.

The two RICH detectors are essential for particle identification at LHCb. Each uses photon detectors to measures the Cherenkov radiation emitted by particles passing through aerogel and C_4F_{10} gas at speeds greater than the speed of light in those substances (but never greater than the speed of light in vacuum). Cherenkov radiation is emitted in a cone; a mirror reflects it onto the sensor, where it leaves a circular image, the radius of which is proportional to the angle of emission. The RICH1 detector, placed next to VELO, is optimized for low-momentum particles, while the RICH2 detector, placed in front of the calorimeters, is optimized for high [27].

The tracking stations TT and T1-T3 are silicon microstrip detectors measuring charged particle momentum for precise unstable particle mass resolution and track direction for particle identification in RICH detectors. The TT station, placed between RICH1 and the magnet, assigns transverse momentum information to large-impact parameter tracks and is also used in offline analysis for long-lived neutral particle reconstruction [27] [28].

SPD, PS, ECAL, and HCAL together comprise the calorimeter system. This system identifies and measures the energies and positions of hadrons, electrons, and photons. Its other essential function is to detect photons with enough precision to enable reconstruction of B-decay channels containing prompt photons or neutral pions π^0. The electromagnetic calorimeter consists of lead sheets interspersed with scintillator plates in a “shashlik” layout. The hadronic calorimeter is an iron/scintillating tile readout. The SPD/PS system is made from a lead converter plate that is sandwiched between two layers of scintillator pads characterized by being relatively large in number, compact, and fast. The SPD determines whether particles are charged or neutral, while indicates the electromagnetic character of the particle [29].

The muon stations M1-M5 capture another particle type involved in b quark interactions [30]. Muons are present in the final states of many B-decays and play a major role in CP violation. The muon stations are the most shielded detector subsystem of LHCb and receive fewer particles than the others. Each station contains chambers filled with a combination of three gases: carbon dioxide, argon, and tetrafluoromethane. The passing muons react with this mixture, and wire electrodes detect the results.

2.2.1 VELO Pixel

In the upgrade following Long Shutdown 2, scheduled for 2019, VELO will be upgraded to a pixel module-based design capable of 40 MHz readout [31]. This will be achieved by completely replacing its sensors and electronics. The current system has radial and azimuthal angle-measuring strip sensors arranged perpendicularly to the beam along the length of about one metre. The new system will replace these with hybrid pixel sensors in grid arrangements. This change will call for new algorithms for processing the output.

VELO Pixel's conceptual layout is shown in Figure 3. VELO Pixel will consist of 52 modules placed along the z-axis, each featuring 12 radiation-hard VeloPix ASIC chips with 256x256-pixel matrices. Each pixel is a 55-µm square silicon sensor. The chips are grouped by rows of three, each bump-bonded to a single sensor to form a tile. Four tiles are arranged around the LHCb beam pipe in an L shape, so that two tiles are read out along the x-axis and two along the y-axis. Sensors on one side overlap with sensors on the other in order to prevent loss of angled tracks.

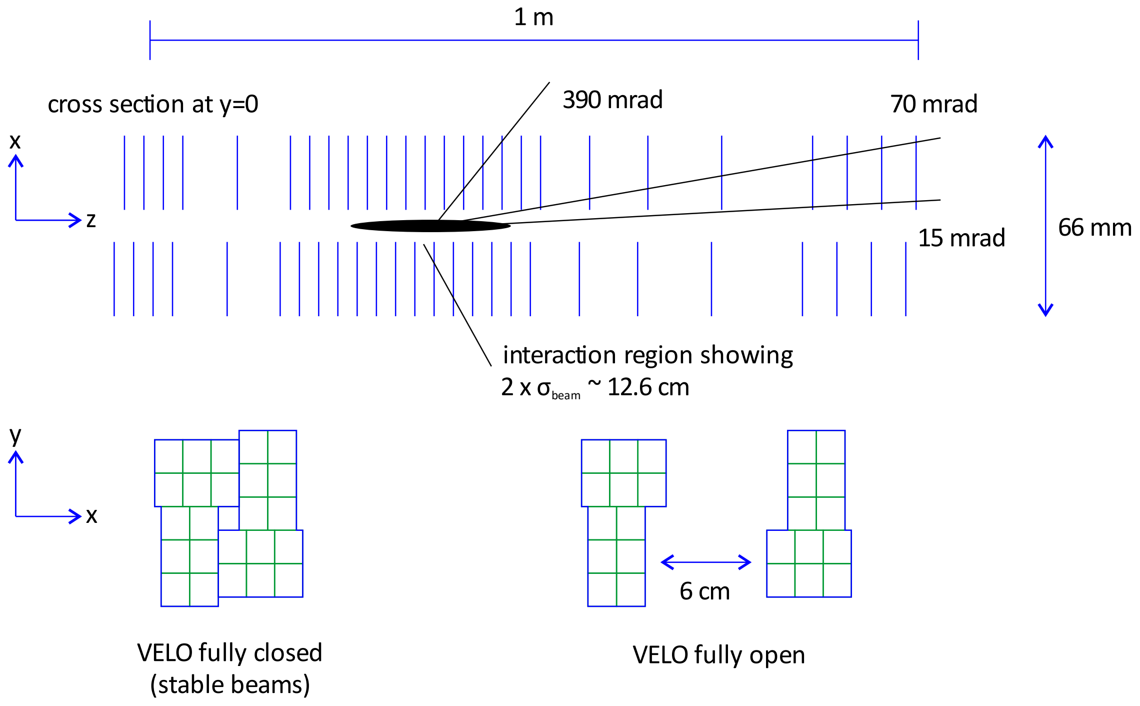

Figure 3 Schematic layout of the upgraded VELO.

Particle collisions occur within the interaction region. Most new particles are created in close proximity to it, and VELO's primary task is to find where each particle is created, so that the other subdetectors can then accurately determine its trajectory. Due to the fact that VELO surrounds the interaction region, it also adds partial information about tracks falling outside the subdetector's acceptance region, including the ones that fly off in the opposite direction.

2.3 LHCb Computing

The LHC fires particles in bursts, called "bunches". These bunches move along the circular beam in opposite directions and are timed very precisely to cross and collide at certain points where their collisions are observed. Forty million bunches are fired each second — in other words, the detectors have a 40 MHz bunch-crossing rate. One of the main challenges of the current experiment is to reduce this rate to a rate acceptable for storage. The system responsible for selecting the small fraction of "interesting" events is called a trigger. The trigger works in stages by first partially and then fully reconstructing events based on raw data coming from the subdetectors. This task requires a powerful and sophisticated computer.

LHCb's computations are split into Online, tasked with the real-time analysis of the detector's output, and Offline, tasked with stored data analysis. The Online system consists of the ECS (**e**xperiment **c**ontrol **s**ystem), the TFC (**t**iming and **f**ast **c**ontrol), OIT (**o**nline **IT** infrastructure), and DAQ (**d**ata **acq**uisition). Its fundamental goal is to satisfy the needs of the physics program [32].

The ECS is a uniform control system in charge of configuring, monitoring, and controlling both DAQ and various detector elements. Its physical backbone is a large private local-area network with over 100 servers, where all the essential services, such as domain control and user accounts, are mirrored from the CERN domain to ensure independent operation. The most important of its functions is behavioural modeling of all the software and hardware devices in the system as finite state machines, which allows a single operator to run the entire experiment, assisted by a second person monitoring data quality.

The TFC transmits synchronous, fast signals, such as trigger decisions and the clock. It is physically implemented around the optical, radiation-hardened TTC (**t**iming, **t**rigger, and **c**ontrol) technology developed for first-generation LHC experiments.

OIT consists of general-purpose work servers, terminal stations, databases, and all the services required for running the Online system.

The basic design of LHCb's DAQ is strongly influenced by previous experience building and operating DELPHI, as well as ALEPH (**a**pparatus for **LE**P **ph**ysics). Its design allows it to maintain very high running efficiencies, to adapt the system to changing needs, and to operate under special running modes.

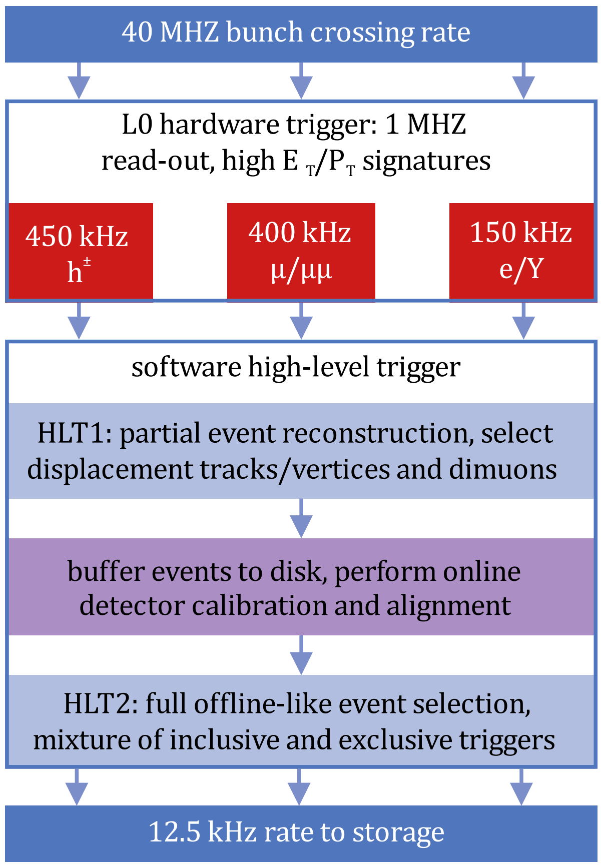
40 MHZ bunch crossing rate
L0 hardware trigger: 1 MHZ
read-out, high E_T/P_T signatures
450 kHz
$h^{\pm}$
400 kHz
μ/μμ
150 kHz
e/Y
software high-level trigger
HLT1: partial event reconstruction, select displacement tracks/vertices and dimuons
buffer events to disk, perform online detector calibration and alignment
HLT2: full offline-like event selection, mixture of inclusive and exclusive triggers
12.5 kHz rate to storage

Figure 4 LHCb 2015 trigger diagram **[33]**

In the DAQ, the calorimeter, the muon system, and the VELO pile-up sensors feed into the hardware L0 trigger at 40 MHz. The L0 trigger reduces the rate to 1 MHz and passes the data to the event builder. The event builder aggregates data from the different subdetectors into event objects, averaging 100 kB each, and passes them into the event filter computer farm. There, the software HLT (**h**igh-**l**evel **t**rigger) applies sophisticated pattern recognition algorithms and reduces the event rate to about 12.5 kHz, acceptable for permanent storage.

The L0 trigger's purpose is to reduce the rate of crossings below the limit at which HLT can process them. It used to be implemented in the readout electronics, together with timing and control, which required it to reliably perform within strict time constraints.

Implementing the L0 trigger in hardware comes at a cost to the physics program at LHCb by discarding more than half of the interesting events from *B* hadron to hadron decay. Moreover, this setup lacks flexibility to adapt to changes in the experiment. It is planned that after 2017, the detector will be upgraded, and a new, entirely software-based, LLT (**l**ow-**l**evel **t**rigger) will take place of L0. LLT would operate on data from the whole detector at every bunch crossing and throttle input to HLT.

The software HLT runs on the off-the-shelf CPUs of the EFF (**e**vent **f**ilter **f**arm). As of October 2015, the EFF is comprised of 1,820 nodes with 27,040 physical CPU cores among them [34]. The software HLT is split into two stages: HLT1 and HLT2.

There are two important points in a particle decay track: the primary vertex and the secondary vertex. The primary vertex is the interaction point location, and the secondary vertex is the location of the decay. It is crucial that they are reconstructed precisely.

HLT1 reconstructs particles in the VELO and determines the position of the event's primary vertex. It reduces the rate to a level that allows tracking of all VELO tracks across the other subdetectors. HLT2 searches for secondary vertices and applies decay length and mass filters to reduce the rate to a level at which events can be written to storage.

3 LHCb software infrastructure

The previous chapter has introduced the hardware setup of the LHCb experiment and gave a high-level overview of LHCb Computing. The current chapter complements the preceding by diving in deeper and describing the software environment in which the data produced by the LHCb experiment is processed.

This chapter tells how the requirements of the experiment have shaped its software infrastructure. It introduces the main software framework supporting the rest of the infrastructure and explains the principles of software organization in the LHCb experiment. The final section is devoted to an effort at creation of a new version of Gaudi designed to be much better able to use modern multicore processors efficiently.

3.1 Requirements

The LHCb Computing project [35] provides the software infrastructure for all the software data-processing applications, from the high-level trigger to physics analysis. It is also in charge of coordinating processing and storage resources, as well as providing all the tools needed for their management.

Typically, fewer than 10^{-4} of the b-quark particles produced in collisions are of interest for CP violation studies. This is why a very sophisticated trigger is needed. The data-processing chain takes raw data, selects interesting events according to elaborate criteria, and reconstructs the chosen collision events for analysis by physicists. LHCb's measurements require a very high precision with strict control over any systematic errors, and the system is designed to handle amounts of data as large as 20 billion events a year, totalling petabytes. A large fraction of the accepted collision events is used for precise calibration and analysis of the detector.

The software architecture is designed to adapt to changes in requirements and technology over the experiment's lifetime expected to be on the order of 20 years. For this reason, it adapts an architecture-centric approach. A high level of standardization allows the same algorithms to run both in the Online data centre and the physicists' laptops.

The software is built around a comprehensive object-oriented framework called Gaudi [36] [37]. LHCb reconstruction (Brunel), the trigger applications (HLT), the analysis (DaVinci) package, the digitization (Boole) together with the simulation application (Gauss), and the event and detector visualization program (Panoramix) all work within the Gaudi framework [38]. Reconstruction is the most performance-sensitive task, so our interest is with Brunel (B Reconstruction, UNderstanding Events in Lhcb) [39] [40].

3.2 Gaudi

LHCb's grand software framework is called Gaudi [36] [37]. It hosts a wide range of physics data processing applications; from simulation to reconstruction and analysis, all of these applications are built as Gaudi components. Experiment-specific software, such as the Event Model [41], which describes the event data classes and their relationships, and the Detector Description [42], which provides centralized access to technical information about the detector, is provided within the framework as core software components. The framework, together with these services and applications, constitutes the complete LHCb software system.

LHCb draws a clear line between data and algorithms. Data consists of mathematical and physical quantities, such as vectors, hits, and momenta; it is stored in *DataObject* containers. *Algorithms* have well-defined inputs and outputs and serve to manipulate these data containers.

The data flow between algorithms is organized via the transient store. The transient store is presented to an algorithm as generic data storage, insulating it from the persistency implementation details. Algorithms communicate with each other by reading data objects from the transient stores, creating new ones and publishing them to the store to be read by others. An algorithm does not need to know which other algorithm has produced any piece of data, it just needs to find it in a well-defined location and in a well-defined state.

The data manipulated by algorithms is distributed over three transient stores, based on the nature of the data and its lifetime. The event data that is only valid while a particular event is processed is put into the *transient event store*. The detector data, including the detector's description, geometry, and calibration, generally has a lifetime spanning many events and is stored in the *transient detector store*. Finally, the statistical data generally lives from the beginning to the end of the whole sequence of algorithms, and is stored in the *transient histogram store*. The detectors behave slightly differently, but they have many things in common and are all based on the same Gaudi component.

Another type of component is a *service*. Services are there to free the algorithm developer from some of the technicalities of making everything work and let him concentrate on the physics content of his algorithm. A service may be shared among several algorithms. A service may be used only for some events or several times for the same events. The same service could be configured differently by different users. As a prominent example of a service, there are one for managing each transient store, offering algorithms simplified data access; there are persistency services, needed to populate transient stores from persistent data and mirror transient store data to persistent storage; there is a job options service, a message service, an algorithm factory, and others.

Gaudi makes it possible to set up pipelines from scripts. A script, usually written in Python [43], combines and configures different algorithms and services, as well as sets up the data sources and describes the order of execution of all the components in the pipeline.

To facilitate component configuration, Gaudi components transition between the four states shown in Figure 5. An algorithm or service begins in the offline state. Configuration is normally managed by the framework by setting properties exposed by the component to scripted values. Initialization is performed once before running the component and is the time at which the component should prepare its data structures for run time. Once it is initialized, a component could be run multiple times.

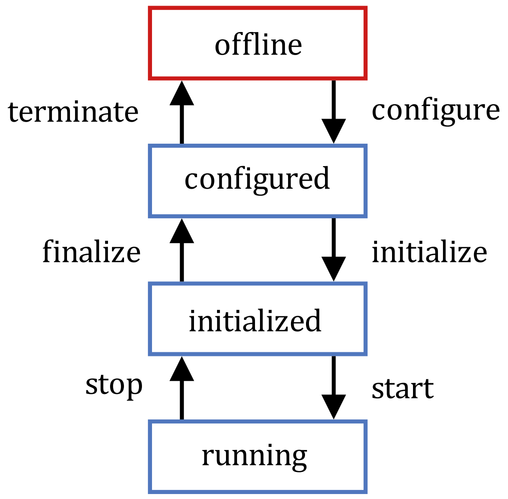

Figure 5 Gaudi object state diagram. The terminal state is marked red.

Gaudi also provides some additional services to aid performance evaluation, such as the times between *start* and *stop* event pairs for a component. We will use this information for measuring CPU algorithm performance.

Gaudi adds a small amount of overhead to run the algorithm. When you start a Gaudi instance, it takes the path to a Python script file as a parameter and bootstraps the application by loading the *application manager* component. The application manager is in charge of creating an initializing a minimal set of basic and essential services before control is given to the *event loop manager*, which invokes algorithms from the top-level algorithm list for each physics events. The algorithms and the events processed by the event loop manager are set up in the Python script.

The event loop manager invokes its algorithms sequentially in the order they are specified. This very basic scheduling is insufficient for use cases, such as event filtering and conditional execution — a shortcoming remedied by specialized scheduling algorithms, such as sequencers and pre-scalers, which invoke other algorithms, also configured through scripting. All of this functionality is single-threaded, and so occupies memory, but does not run at the same time as the physics algorithms. Therefore the algorithm timings provided by Gaudi are not significantly affected by its overhead.

3.3 Software organization

Many of the LHC experiment algorithm developers are participating physicists rather than professional programmers. In such a team, it is vital to keep the amount of boilerplate code to a minimum, both in the software and in its configuration scripts [44]. This is one of the main guiding principles in the design of its system of organizing the source and the development process.

Developers express their algorithms in text called source code. The source code is turned into executable components by a build system. Build systems provide the means for expressing dependencies between different components, so that the source code in one component can use the functionality of another. LHCb's build system also sets conventions for how the code is organized and how the components are deployed. LHCb has started out using a tool called CMT [45], used by a few high-energy physics experiments, for managing its build system, however this tool is now being phased out in favour of an industry-standard system called CMake [46].

The build system is organized around the concept of *packages* (*subdirectories* in CMake terminology). A package usually contains a single Gaudi component. Packages can have dependencies between each other, so that one component can refer to another. The dependencies and all other information required for building a component is stored in a configuration file.

Projects are collections of packages. The project configuration file is used to set up an environment for the build system. Packages can easily be added or removed from a project, so that it is possible to reorganize them by moving between projects without having to significantly modify the configuration files. It is also possible for a project to override a package from a project it depends on to provide an urgent bug fix without having to wait for a new release or to test out a new implementation. LHCb's software consists of over 700 actively developed packages grouped into about 30 projects. Brunel, mentioned in section 3.1 is an example of a project.

3.4 Event data model

The event data model is the data structure that describes the LHCb event data [47]. Event data my come from the detector or from simulation. Simulated events could contain additional information, such as simulated tracks for testing tracking algorithms. As mentioned in section 3.2 , the events are accessed through the transient event store by referencing well-defined locations. The typical contents of an event are described in Table 1. Different types of data are located in branches in the "event" directory node. Each branch can have further sub-branches, such as */Event/Raw/ECAL* and */Event/Trig/L0*.

Location	**Content**
/Event/Gen	Simulated events.
/Event/MC	Simulated tracks.
/Event/Raw	Raw output from the detector or simulation.
/Event/Trig	Simulated trigger output.
/Event/Rec	Reconstructed event.
/Event/Phys	Physics analysis.

Table 1 Event data model organization.

Simulated events are produced by the Gauss application [48]. The simulation is performed in two independent phases: event generation and track simulation. The event generation phase accurately imitates particle collisions under given running conditions. The track simulation phase is performed with the help of the Geant4 toolkit [49] [50] and an LHCb geometry description. Geometry of the subdetectors, which were described in section 2.2, is known for the ideal detector as well as for the real hardware, where the positioning was obtained with the help of a geometrical survey.

3.5 Gaudi Hive

There is an ongoing effort to add support for parallelism inside the Gaudi framework, called Gaudi Hive [51] [52]. Normally, multiple Gaudi instances run concurrently, each processing one event at a time; this is known as event-level parallelism. Gaudi Hive extends this by adding in-process *event-level parallelism* and *task-level parallelism*, where the work is divided into tasks and multiple workers execute tasks over the same data. The benefits are much reduced memory consumption, reduced number of required resources. The extension is designed to require minimal modification of the existing algorithms, but the changes that make algorithms more suitable for task parallelism also tend to improve code quality and make them more reliable and maintainable.

Having taken up the challenge of concurrent execution of algorithms that were not designed for concurrency, Gaudi Hive has to deal with multiple algorithms contending for the same resource, such as the output stream for printing messages to the screen. Contention between algorithms has to be accounted for, and so algorithms in Gaudi Hive form a DAG (directed acyclic graph), which allows the framework to know when the next algorithm should be launched and execute multiple independent algorithms at the same time. Different strategies can be chosen for using the DAG to choose algorithms. The strategy to use is specified via a configuration script. Figure 6 shows three different strategies, and detailed explanations follow.

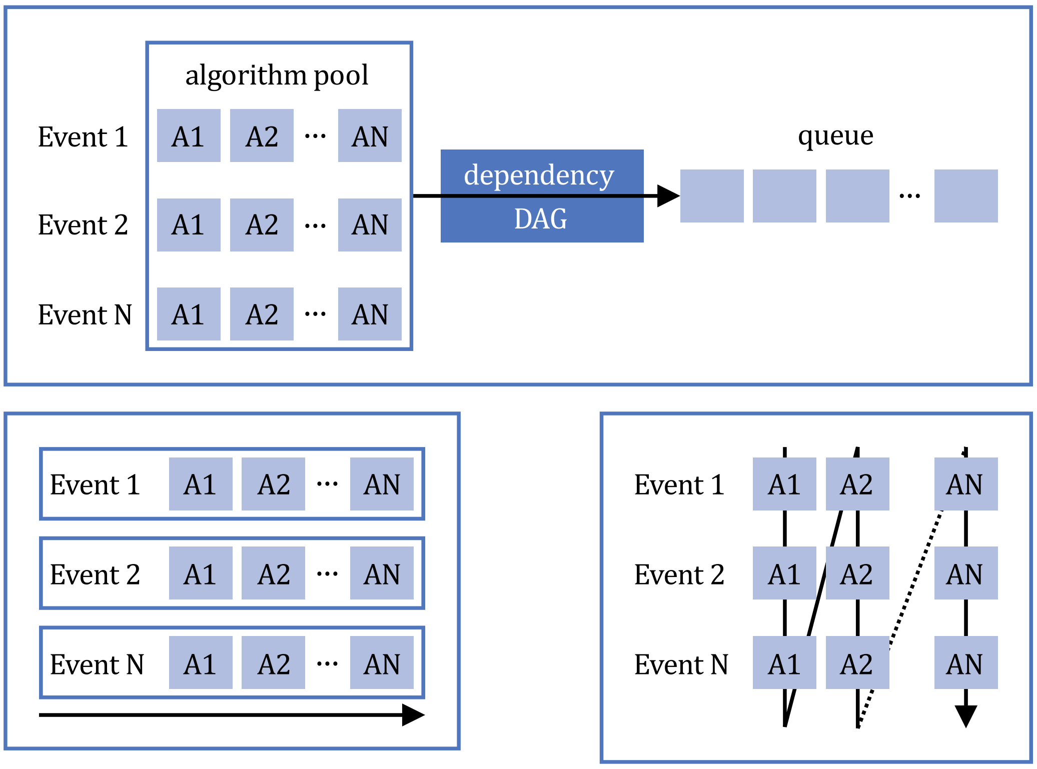

Figure 6 Gaudi Hive scheduling strategies. Forward scheduling at the top, parallel sequential on bottom left, and round robin on bottom right.

The *forward scheduling* strategy schedules an algorithm as soon as its data dependencies are available. In order to perform well, this strategy requires that all the data dependencies are known and that multiple events are processed simultaneously. Forward scheduling is immune to deadlocks and fits well with many-core and heterogeneous systems. It is a natural choice to support offloading of computations, as it allows the CPU to continue working while some of the algorithms wait for external accelerator hardware.

The *parallel sequential scheduling* strategy runs several sequences of algorithms for multiple events. Compared to the normal procedure of running multiple Gaudi processes, this strategy uses much less memory and simplifies input and output management. This strategy does not require dependencies between algorithms to be specified, but it is less flexible than forward scheduling and benefits less from computation offloading.

The *round-robin scheduling strategy* also runs several sequences of algorithms for multiple events, but it runs one algorithm at a time and gives a time slot to each event. This strategy is designed to gain the performance benefits of being cache-friendly, but it adds the additional requirement that all the algorithm sequences are the same. This scheduling scheme can be used without any changes to current algorithms.

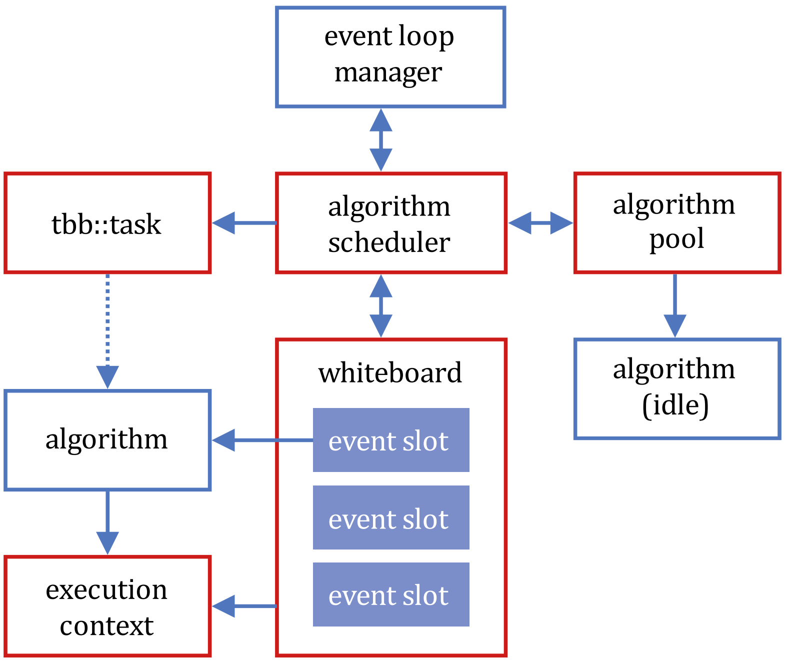

Figure 7 Gaudi Hive design diagram. The new components are highlighted in red. Arrows indicate the direction of control. The dotted arrow from tbb::task to algorithm indicates indirect control through lambda functions.

Figure 7 illustrates how Gaudi Hive fits into the current framework. Whereas normally the event loop manager simply executes a series of algorithms listed in a configuration file, Gaudi Hive adds several intermediate components. Gaudi Hive uses the Intel TBB (**t**hread **b**uilding **b**locks) library for modelling and scheduling tasks.

In Gaudi Hive, the *event loop manager* loops over the events and hands them over to the *algorithm scheduler*. The algorithm scheduler requests algorithm instances from the *algorithm pool* — the entity responsible for containing and coordinating them — packages them into tasks, and submits them to the TBB runtime.

The *whiteboard* is necessary for achieving event-level parallelism. It acts as a façade for multiple event stores by implementing the event store algorithm itself. The calls to the whiteboard are transparently forwarded to the appropriate event stores using information stored in the *execution context*. Whiteboard also implements some additional functions, such as newly added data object bookkeeping for algorithm scheduling.

Figure 8 shows the state of the Gaudi Hive queues while scheduling 263 tasks using 8 and 20 threads. The ideal scheduler's diagram would have all bars evenly distributed under the red line, corresponding to constant full thread utilization.

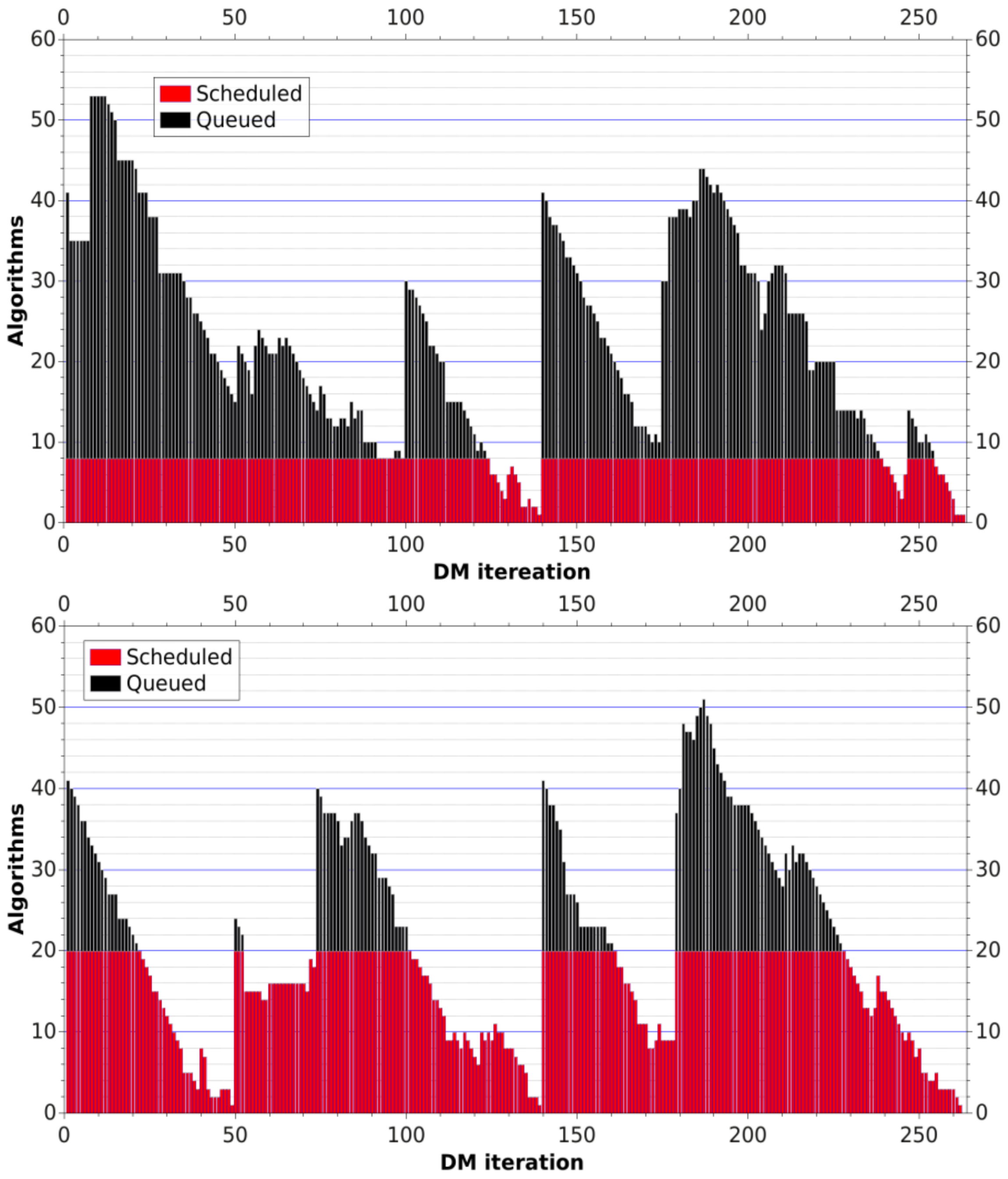

Figure 8 Gaudi Hive decision-making iterations for scheduling 263 tasks using 8 threads (top) and 20 threads (bottom) **[53]**.

Gaudi Hive serves as an advanced example for how to extend the Gaudi infrastructure for concurrency without tearing down what has been built before. It is also an important consideration for computation offloading, since any such service will likely have to interact with Gaudi Hive in the future.

4 The need for speed

With the hardware setup established, the current chapter explains why more performance is needed and why qualitative, rather than quantitative change in hardware and algorithms is desirable.

The main change precipitating the current research is LHC's high-luminosity upgrade that will increase luminosity by an order of magnitude from 2017 onwards. LHCb plans major changes in this upgrade, changes that will place sharply increased requirements on the computational capacity of the LHCb compute farm. This chapter discusses the upgrade and the main computational alternatives being considered for coping with the new demands.

4.1 High-luminosity upgrade

One of the main performance parameter of the LHC is its luminosity — it refers to the number of particles passing through the accelerator per unit time; the more particles in motion, the more collisions we expect to happen at the detector sites. LHCb has been designed to operate with an average $2 \times 10^{32} cm^{-2} s^{-1}$ luminosity, but as of 2012 it has exceeded it by a factor of two. Still, with LHC providing a luminosity of $10^{34} cm^{-2} s^{-1}$, LHCb has to operate with slightly defocused beams to reduce it to a manageable level. Because LHCb is a specialized detector, unlike the more general-purpose ATLAS and CMS, even this mode of operation provides large numbers of interesting collisions.

One reason for limiting luminosity at LHCb is to reduce radiation damage on the detector components. As materials and processes improve, radiation tolerance can be increased; however, the other reason for limiting luminosity is the 1 MHz detector readout rate and the limited discriminative power of the L0 trigger. New computational challenges will arise when LLT replaces L0.

Increasing the readout rate by the data centre from 1 MHz to 40 MHz increases the demand on hardware by a factor of 40. A factor of 10 could come from improvements in processor power by the time of the upgrade. Another factor of 4 could come from increasing the number of servers. However, raising the luminosity would also increase the average number of hits per event, and some of the algorithms take time proportional to its square or cube [54]. If LHCb luminosity were to double, a quadratic-complexity algorithm would require 4 times more computational power.

LHCb's datacenter is located 100 meters underground and is limited in space, cooling, and power consumption, restricting possible expansion, but the most important limitation is made by the fixed upgrade budget. It is therefore necessary to look at better ways of performing computation to keep up with increasing luminosity.

4.2 Computation alternatives

Over the first 30 of the past 40 years, CPUs gained performance from higher clock speeds, more effective instruction execution, and better caches. However, around 2003, chip clock speed and power consumption reached their peaks, while the number of transistors per chip continued to increase exponentially in accordance with Moore's Law [55]. CPU designers had to look to new ways of spending their transistor budgets to increase software performance.

Some algorithms are inherently serial: each step has to be made in turn after the previous one. Such algorithms get little benefit from the technological advances of the past decade. Other algorithms have parts that can be executed in parallel, independently of each other. Amdahl's law [56] relates the proportion of the algorithm that has to be executed serially B and the number of threads n available for execution of the parallel portion. Denoting the time it takes for an algorithm to execute as T, the speedup is equal to:

$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{B + \frac{1}{n}(1 - B)}$$

We distinguish between task-parallel algorithms and data-parallel algorithms. Task-parallel algorithms perform multiple concurrent sequences of operations on the same (or different) data. In data-parallel algorithms, the data is divided between the differences threads; in particular, SIMD (**s**ingle **i**nstruction, **m**ultiple **d**ata) instructions execute identical concurrent sequences of operations on separate pieces of data.

LHCb's needs are somewhat different from those of the other experiments. Events produced at LHCb are relatively small. Furthermore, a certain degree of latency can be tolerated in event processing. This makes LHCb events naturally suitable for data-parallel computation on GPUs. We accept that not every algorithm can be rewritten for GPU execution. Amdahl's Law tells us the overall speedup, given a proportion of algorithms in an event-processing chain rewritten for GPUs.

4.3 Intel Xeon Phi

The LHCb EFF runs mainly on Intel Xeon processors. Xeon is a family of CPUs intended for workstations and servers with high performance and reliability requirements. Xeon CPUs have multi-socket capabilities, high numbers of cores, and ECC (**e**rror-**c**orrecting **c**ode) memory support.

Intel Xeon Phi is a family of coprocessors using Intel's MIC (**m**any **i**ntegrated **c**ores) architecture that incorporates Intel's past research into GPUs. A Xeon Phi processor combines many cores augmented with VPUs (**v**ector **p**rocessing **u**nits) in a single chip. Each coprocessor is equipped with its own high-bandwidth RAM (**r**andom-**a**ccess **m**emory) and acts as a separate Linux node communicating with the host Xeon CPU via the TCP/IP protocol. This setup is designed to provide high throughput per watt for large data sets and highly parallel algorithms.

Intel provides a suite of development tools for working with the Xeon Phi, including C++ and Fortran compilers, debuggers, and profilers. These tools would be familiar to those who use them for other Intel's processors; however, support for free common tools typically used in LHCb's software development environment is limited.

Intel recommends a data-parallel approach to algorithm design for the Xeon Phi. Data-parallel algorithms distribute input across available cores; this makes the data readily available for processing using VPUs and minimizes the amount of memory used per core. This design also requires less communication between cores.

4.4 GPU programming

4.4.1 GPGPU

The term "GPU" was first coined by NVIDIA in 1999 during the launch of its GeForce 256 video card, but the first graphics processing units appeared in the early 1980s for professional applications, such as computer-assisted design. The IBM Professional Graphics Controller was released in 1984 with a price tag of $4,290 as one of the first PC GPUs. It came with its own processor and memory.

In mid-1990s, several companies started producing consumer-level GPUs for speeding up video game rendering. For the first decade of their development, GPUs were forced to use the PCI bus, developed by Intel for attaching peripherals. This bus was slow and shared bandwidth among all the devices connected to it; sending data to a graphics card through PCI was much faster than receiving data from it. Thus, the entire pipeline for consumer graphics cards was narrow and one-directional. Due to these constraints, graphics cards were severely limited in their ability to specialize: since sending data back to the CPU was so expensive, they had to perform many types of the calculations that the CPU was capable of.

In a typical pipeline, a GPU would load lists of 3D polygons and textures from main computer memory, apply a set of fixed-pipeline functions and a camera projection matrix, then output a raster image to the display. This required performing the same sets of instructions for every polygon vertex and every texture pixel. As games became more demanding, they came to process scenes with tens and hundreds of thousands of polygons, screen resolutions counted in the millions of pixels, and antialiasing techniques asked for more than one sample from each of those pixels. All the while the computation loads increased, the typical rates of 30-60 frames per second allowed GPUs to be tolerant with respect to latency. This would define GPU architecture.

The constraints imposed on the GPUs forced them to gain more and more features of general processors. The pipelines were broken into increasingly specialized intermediate stages that could be controlled by the CPU. Eventually, GPUs moved past fixed-pipeline transforms and let developers load custom programs, called *shaders*. This then gave rise to the use of GPUs for general-purpose computing — GPGPU.

Without the burden of having to achieve high performance on legacy code, Graphics Processing Unit (GPU) manufacturers have been free to seek optimal combinations of instruction sets, hardware, and software architectures. Instead of maximizing serial performance, they opted to create large chips operating under the lowest sustainable voltage. With as many as thousands simultaneous threads and vectorized instructions, this approach enables impressive data throughput with low power consumption for massively data-parallel algorithms.

Early GPGPU programming was complicated and required significant effort to subvert graphics application programming interfaces and shader programming languages for general-purpose computation. It was also limiting that GPUs were heavily optimized for single-precision floating point mathematics, whereas many scientific applications often required double precision.

Eventually, GPU producers and API developers stepped in and offered their support to the GPGPU scene. OpenCL (**open c**omputing **l**anguage) [57], developed by the Kronos group, became an open-standards framework for writing programs running on heterogeneous platforms, including common CPU and GPU processors. OpenCL provides an API for communication between software and hardware, as well as reference compiler and library implementations and a conformance testing program for third-party implementations.

Game-oriented graphics APIs DirectX 12 from Microsoft [58] and Metal from Apple [59] also make it easier to take advantage of the general-purpose computation power of modern GPUs. Nvidia — the leading discrete graphics card manufacturer —developed its own proprietary platform and GPUs dedicated to general-purpose computation, called CUDA and Tesla. Tesla processors are optimized for double-precision math desired by scientists.

4.4.2 CUDA

CUDA is a parallel computing platform that Nvidia created to make it easier to write general-purpose computing programs for Nvidia's hardware. The challenge is to allow the programmer to write high-performance code in a familiar language and have it be portable across many hardware models and generations.

CUDA achieves transparent scaling over many cores by using the concept of a kernel. A kernel is a function that simultaneously executes over an array of threads. This function can be written in a high-level language, like C++. The GPU executes one kernel at a time using all the cores it has available.

Threads are organized into blocks, and blocks are automatically distributed over GPU cores. Blocks mark communication boundaries; threads in the same blocks can communicate with each other using shared memory, but not with threads in other blocks. The programmer chooses how many blocks and how many threads per blocks are needed.

There is also an intermediate layer between the C++ code and raw GPU instructions. C++ code is first compiled to PTX ISA (parallel thread execution virtual machine and instruction set architecture) code. This code plays a similar role to LLVM bitcode, Java bytecode, and .NET CL — it provides a stable platform for CUDA programs and is translated to instructions for the specific target GPU before execution.

4.4.3 GPUDirect

GPUs process data from their own private memory stores. In order to pass data from CPU memory to a GPU kernel, it first has to be copied to the GPU. If one GPU needs to copy data to another, or if a CPU accesses a GPU across a network, the data would normally need to be copied to an intermediary CPU first. This copying adds significant overhead.

Nvidia has developed the GPUDirect technology for use with its supporting GPUs that allows data to be exchanged between GPUs with fewer intermediaries. In this case a memory buffer is "pinned" for a given GPU and then accessed directly via PCIe or Infiniband.

GPUDirect has gone through three iterations. In v1, it reduced the number of buffers for communication between GPUs across a network. In v2, it added the ability for several GPUs on a single node to exchange data directly. Finally, in v3, it added support for RDMA (**r**emote **d**irect **m**emory **a**ccess), allowing data exchange without going through host memory at all.

Infiniband is a type of high-performance low-latency interconnection with RDMA support. It was designed for high-performance computing and is commonly used in supercomputers. It can be a good solution in high-energy physics, where low latency is required for detector readout [60] [61].

GPUDirect does not come without a cost, however. Memory pinning is an expensive operation that can take up to milliseconds, meaning that pinned buffers should be reused as much as possible. The amount of memory available for pinning is limited: depending on the GPU and the motherboard, it can be as small as 256 MB, with 32 MB of that reserved for internal purposes.

5 State of the art

The fast pace of development in massively parallel hardware has been a subject of interest for many high-energy physics experiments, including the ones at CERN, for some time. Many researchers have been investigating new algorithms optimized for GPUs and the problems of integrating them with existing infrastructure.

The current chapter examines previous work that would be useful in guiding the integration of massively parallel computations at LHCb. It focuses in particular on the work done by the NA62, ALICE, and ATLAS experiments.

5.1 NA62

Other experiments also face difficulties in dealing with high computation loads, feeding an increasing interest in high-energy physics GPU processor applications in recent years. Most of the efforts so far have dealt with the problem of adapting specific algorithms to massively parallel architectures. The NA62 experiment has investigated real-time use of GPUs for its trigger and data acquisition system [3].

The NA62 experiment is part of CERN's SPS particle accelerator. The SPS is a synchrotron-type particle accelerator, 6.9 km in circumference, shown as a circle around ATLAS in Figure 1. Like LHCb, NA62 specializes in tracking rare decay events; it focuses on testing the Standard Model by tracking charged kaon decays. A kaon is a kind of hadron (explain in section 2.2) characterized by its quark composition. The Standard Model predicts that $8.7 \pm 0.7 \times 10^{-11}$ of kaons decay in the particular mode NA62 tracks.

In order to track such rare events, the experiment needs an intense beam, a reliable trigger system, and a lossless DAQ. They are divided into three levels L0-L2. Like LHCb, NA62 uses a hardware L0 trigger for performing preliminary selection in advance of sending the information down to its data centre. The L0 trigger reduces information rate from 10 MHz to 1 MHz by performing partial track reconstruction and selection. The L1 trigger makes decisions based on individual subdetectors. The L2 level makes decisions based on fully reconstructed events at high resolution.

Fast RICH reconstruction, used at both L0 and L1 levels, was ported to the GPU using Nvidia's CUDA framework. Because L0 requires low latency, this aspect was deemed especially important. Special care has been taken to pre-allocate memory and to not process more data than possible within allotted time. The implementation uses multiple levels of parallelism, including multiple event batching. Tests have been carried out on two machines:

Machine 1:	Machine 2:
Intel Xeon E5630	Intel i7 3770
Nvidia Tesla C2050	Nvidia GTX 680
12 GB DDR2	16GB DDR3
PCIe v2	PCIe v3

The latency measurements are summarized in Figure 9. The algorithm succeeded in processing 1000 events in 60 µs, achieving a throughput of 2.6 GB/s, which fits within NA62's requirements.

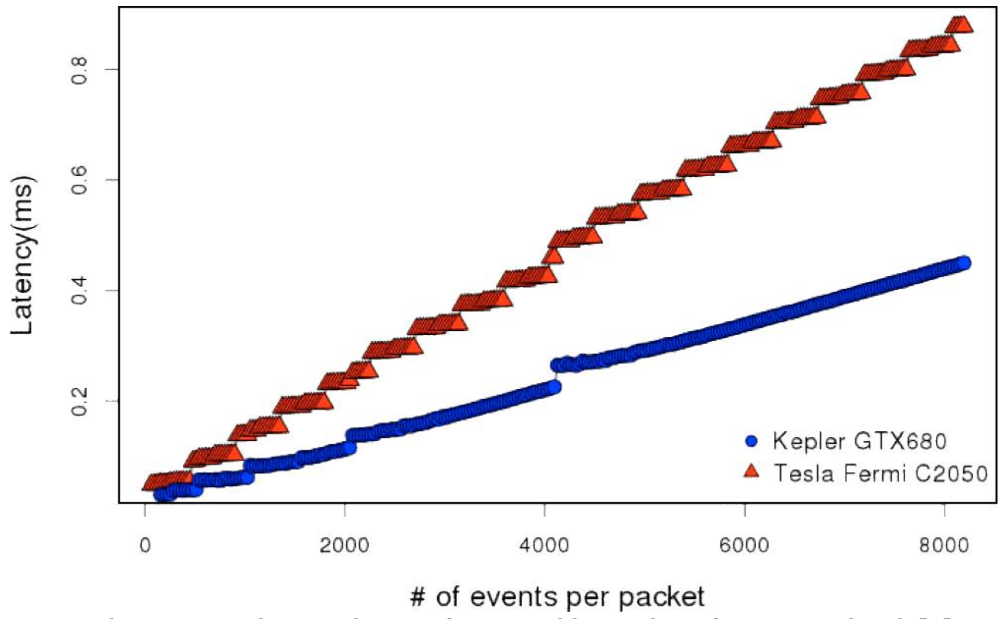

Figure 9 Ring detection on the GPU: latency for a variable number of events per batch **[3]**.

5.2 NA62 NaNet

The NA62 collaboration has also addressed another issue of prime importance for the use of GPUs in latency-sensitive applications, such as the NA62 L0 trigger. They have developed an NIC (**n**etwork **i**nterface **c**ard), called NaNet, capable of copying data directly from the hardware readout boards to GPU memory via NVIDA's GPUDirect technology. The NIC has hardware support for custom and standard network protocols, thus avoiding OS jitter effects and guaranteeing deterministic latency behaviour, while providing maximum throughput. It is also capable of hardware data preprocessing, such as decompression, reformatting, and event fragment merging [62] [63].

The first-generation NaNet NIC, called NaNet-1, is an overhaul of APEnet+ — an earlier GPU-optimized NIC developed in close collaboration with NVIDIA [64] — for real-time data acquisition. NaNet-1 is based on the Stratix IV FPGA (**f**loating-**p**oint **g**ate **a**rray) developer kit. The second-generation NaNet NIC, called NaNet-10, is based on the Stratix V FPGA.

Figure 10 and Figure 11 show latency and throughput measurements for both NaNet generations. In both cases, NaNet-10 shows a large improvement over NaNet-1, while the transfer technology does not make any significant difference. Throughput peaks for message size of 1 KB.

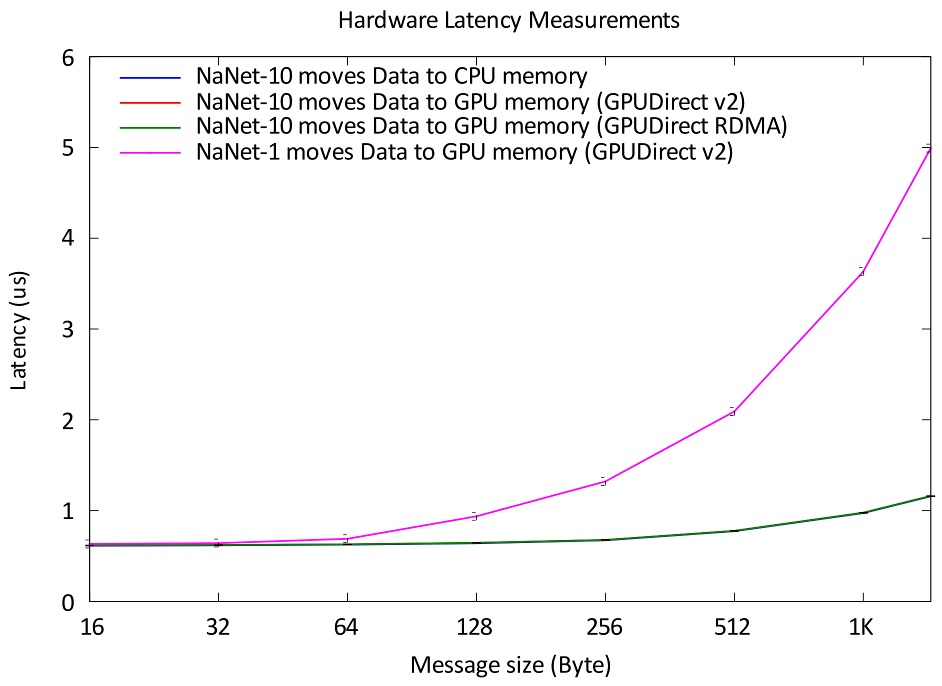

Figure 10 Latency for NaNet-1 and NaNet-10 using different transfer technologies **[63]**.

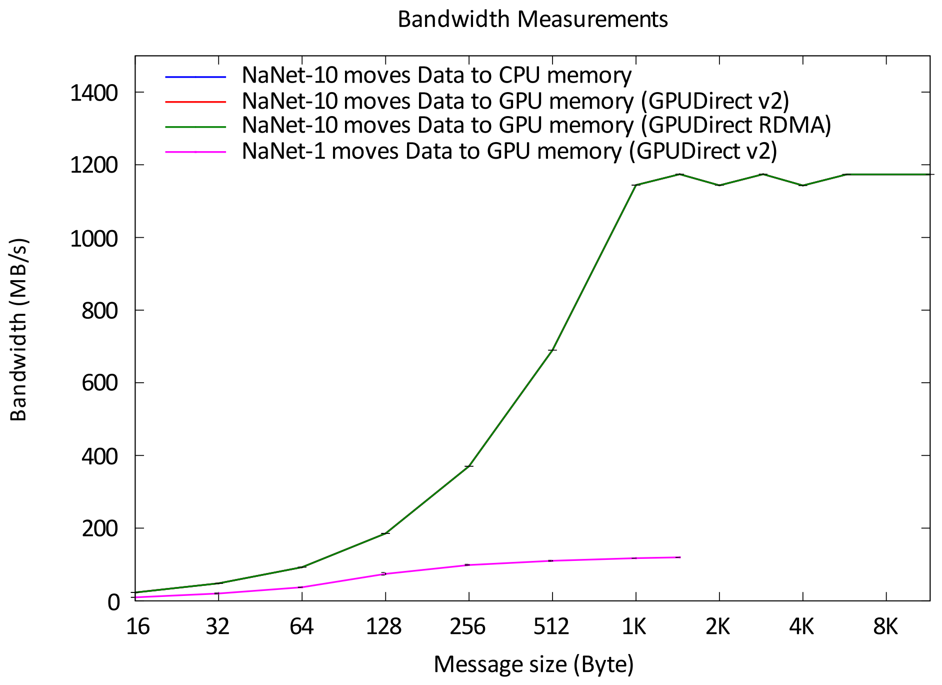

Figure 11 Throughput for NaNet-1 and NaNet-10 using different transfer technologies **[63]**.

In order to test NaNet's performance in a practical application, it was connected to two TEL62 readout boards and a SuperMicro server running an Intel Xeon E5 2620 CPU with an NVIDIA K20c Kepler GPU. The GPU performed multiple-ring RICH event reconstruction on batches of incoming events, called CLOPs (**c**ircular **l**ists **o**f **p**ersistent data-receiving buffers). Simultaneous events coming from different readout boards had to be merged before performing reconstruction.

Figure 12 shows the results of this experiment. Notably, event merging becomes a bottleneck for the algorithm, since it is a largely sequential operation. It could be sped up with a hardware implementation, but a parallel implementation would have to be created in order for the algorithm to be suitable for GPUs.

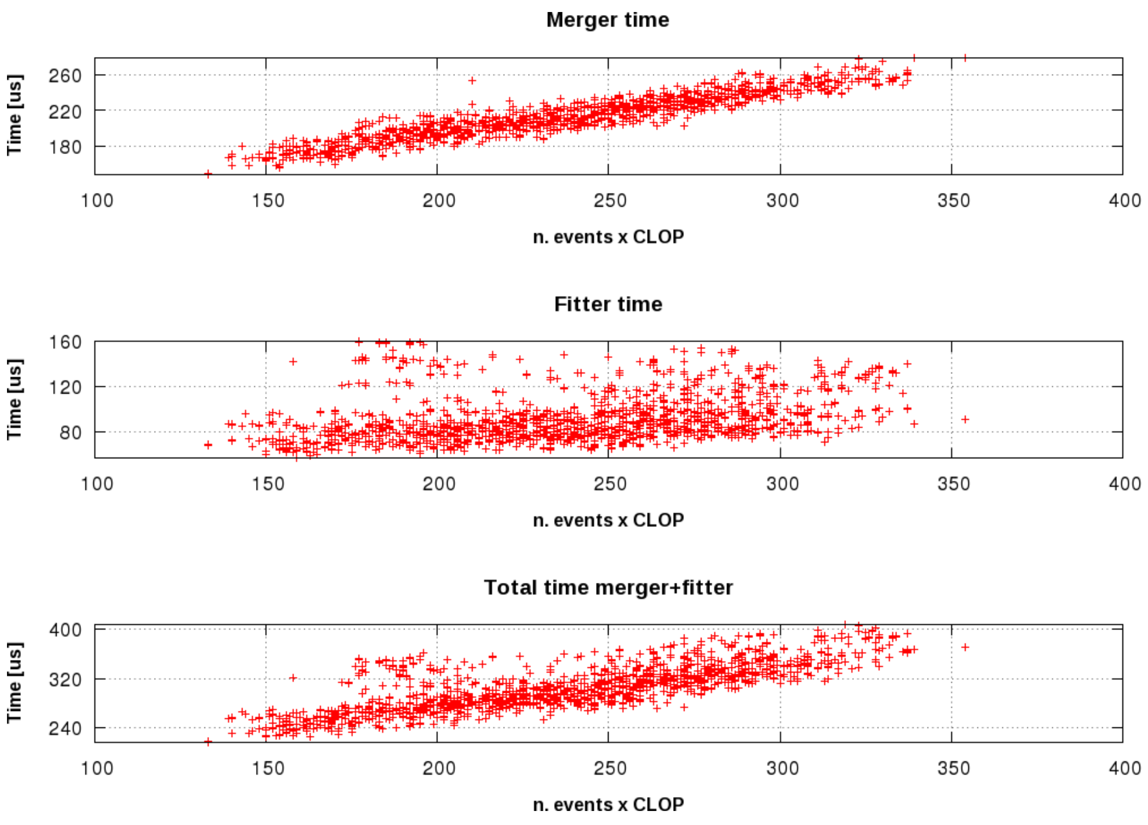

Figure 12 Processing time per event batch for RICH reconstruction using NaNet **[63]**.

5.3 ALICE

Closer to LHCb, ALICE (**a** **l**arge **i**on **c**ollider **e**xperiment) at LHC has also been experimenting with GPU computation. Researchers at ALICE went a step farther than NA62 and investigated integration of GPU algorithms with its existing software infrastructure [4].

ALICE operates a detector at the LHC optimized for heavy ion collisions. Its goal is to study the physics of strongly interacting matter at extreme energy densities, where the formation of a new phase of matter, the quark-gluon plasma, is expected. In this state, quarks would become deconfined and exist as free particles. It is believed that such conditions prevailed in the universe a few microseconds after its formation. Since these conditions cannot be observed astronomically, our only means to study this fundamental state of matter is via the collision of heavy nuclei in laboratory conditions. ALICE also studies proton-proton collisions, both as a comparison with lead-lead collisions and for advancing physics in areas where ALICE is competitive with the other LHC experiments.

The measurements ALICE collects are characterized by very small signal-over-background ration, placing special demands of the trigger. In addition, the detector can be run in several different running modes with rates varying by almost two orders of magnitude. ALICE tries to make optimum use of the data-taking periods by concurrently acquiring data for several observables following different scenarios and giving each a fair share of resources. Like LHCb, ALICE has a hardware trigger and an HLT using general-purpose CPUs; an offline system is used for simulation, reconstruction, calibration, alignment, visualisation, and analysis. ALICE's computer farm consists of about 1,000 off-the-shelf PCs connected to the front end by optical fibers and designed for a 25 GB/s input stream [65].

ALICE's HLT performs event reconstruction, high-level triggering, and data compression. The software architecture is divided into two independently developed parts: the data transportation framework and the data analysis. The analysis framework, called *AliRoot*, can work in online and offline modes. *AliRoot* builds on the ROOT framework, which is a comprehensive set of tools for manipulating, monitoring, and analyzing data in the C++ programming language. It is organized in submodules that can be loaded and unloaded on demand. The framework is responsible for loading and running algorithm modules for online HLT.

ALICE's HLT uses a CA-based (**c**ellular **a**utomata) tracking algorithm for online reconstruction, specially developed with multi-core support in mind. The algorithm could already process proton-proton collisions, but high-rate lead nuclei collisions present a challenge, with up to over 20,000 tracks and several million clusters. This tracking algorithm was ported to the GPU.

Unlike NA62's algorithm, ALICE's GPU-accelerated tracking algorithm processes one event at a time and experienced much difficulty in maintaining high GPU core utilization. Initially, it used under 20% of the GPU; due to the large disparity in track lengths, the GPU, working on many tracks in parallel, would quickly finish the short tracks and then stand idle until the longest tracks were done. The solution was to divide processing into stages: each stage processes a number of tracks for a certain number of steps, and then the remainders are redistributed. An additional pre-filtering pass removes very short tracks even before the event is sent to the GPU. These changes raised GPU utilization to 70%. Figure 13 compares the final GPU port to the original CPU implementation.

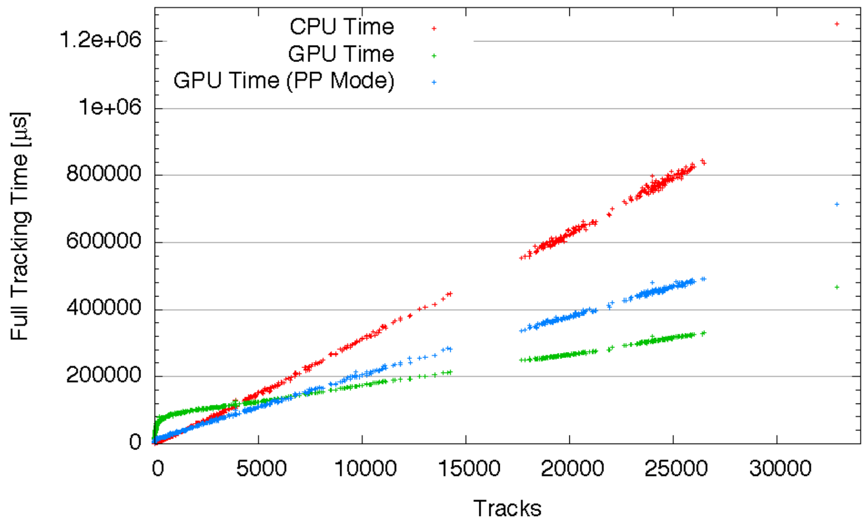

Figure 13 ALICE CA-based tracking: CPU vs GPU performance for different event sizes.

Integration of the GPU tracking algorithm with the existing software necessitated adjustments in the HLT framework, as well as *AliRoot*. They encountered three main problems:

1. Because CUDA is only available on a fraction of cluster nodes, all of the GPU code had to be packaged in a separate library and loaded dynamically as needed.
2. The authors also encountered a mismatch in the threading models used by the HLT framework and CUDA. The reason for the mismatch is that HLT processes events over multiple asynchronous concurrent threads, whereas at the time CUDA was not thread-safe and restricted to single-threaded use. Later versions of CUDA allow access from multiple threads.
3. Finally, the native logging system needed to interpret the source code using ROOT CINT (**C int**erpreter), but could not understand CUDA's extensions. This was solved by giving the logging system a separate non-functional source code file stripped of all the extensions.

5.4 ATLAS

ATLAS (**a** **t**oroidal **L**HC **a**pparatu**s**) is one of the two general-purpose detectors at the LHC. ATLAS has made serious efforts in integration of GPU computation with its software infrastructure. Its design went through stages: first, an RPC (**r**emote **p**rocedure **c**all) architecture [5], and then a higher-level client-server one.

The ATLAS detector is forward-backward symmetric with respect to the interaction point. It provides good charged particle momentum resolution and reconstruction efficiency, very good electromagnetic calorimetry for electron and photon identification and measurement, full-coverage hadronic calorimetry for accurate jet and missing transverse energy measurement, good muon identification and momentum resolution, and highly efficient triggering on low transverse momentum objects with sufficient background rejection.

The ATLAS trigger system is divided into three stages: Level 1, Level 2, and Event Filter. Level 1 is a hardware trigger, while Level 2 and Event Filter together comprise the software-based HLT. The Level 1 trigger uses calorimeter and muon detector information to identify potentially interesting events and ROIs (**r**egions **o**f **i**nterest) within them. ATLAS investigated whether GPUs could improve throughput at increased luminosity under the constraints summarized in Table 2.

Trigger Level	Input Event Rate	Output Event Rate	Latency
Level 1	40 MHz	75 kHz	2.5 μs
Level 2	75 kHz	2 kHz	40 ms
Event Filter	2 kHz	200 Hz	4 s

Table 2 ATLAS trigger event processing rates.

The ATLAS event processing and trigger software uses the Athena framework, which is based on LHCb's Gaudi, described in Section 3.2, with ATLAS-specific enhancements in the event data model and the event generator framework. It was decided that CUDA could not be used directly within Athena, because of Athena's complex build system, high memory usage, and difficult threading, so a client-server system was developed instead.

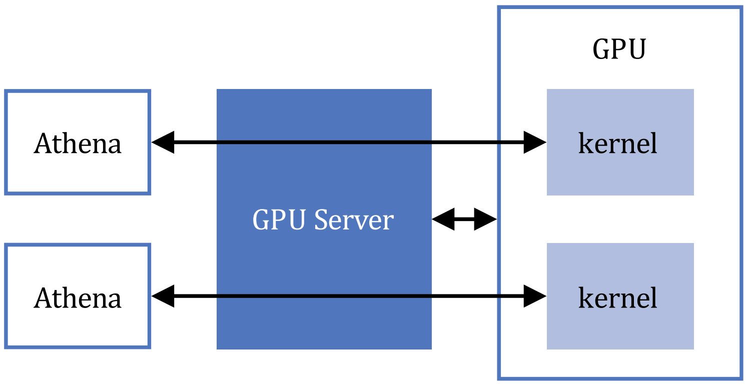

Figure 14 ATLAS GPU Service architecture.

In this approach, the GPU is managed by a separate process, which is responsible for initializing the GPU and for loading onto it the kernels to be made available for use by Athena algorithms. Athena instances communicate with the server. During each communication event, a client submits data addressed to a specific kernel and waits; the server translates it and calls the kernel, then translates the result and returns it back to the client; the client receives the result and resumes activity. The server processes client requests in the order it receives them. This style of communication is commonly called "remote procedure calling" because from the client's point of view, it is similar to directly calling kernels (procedures).

There are some additional costs this method incurs on the developer. The kernel has to be developed using Nvidia's CUDA framework. The kernel has to be compiled along with the source code of the GPU server. The GPU Server also needs custom code for translating the kernel's input and output data.

This approach has some features in common with ALICE's method of packaging CUDA code into dynamically loaded libraries: in both cases GPU kernel calls are made available to regular Athena algorithms as external procedures. Having a separate server offers greater separation between normal Athena pipeline code and CUDA-dependent algorithms. It also has the added benefit of allowing remote calls to CUDA kernels to be made across the network. Finally, hosting all kernels in a separate long-lived process could save on the cost of GPU device setup.

There are significant costs to hosting GPU code in a separate service this way. Every kernel has custom parameters and returns custom data types; addition of new kernels also requires somewhat complicated encoding and decoding changes to the server's request processing loop and to the Athena client. This design also necessitates expensively copying potentially large amounts of data between different processes for sequences of kernel calls when it would be preferable to simply keep this data in GPU memory. A related issue is that the design makes it difficult to reuse resources when calling the same kernel for different events.

5.5 ATLAS APE

The next ATLAS's GPU framework, called APE (**a**ccelerator **p**rocess **e**xtension) was partly inspired by our own effort [66]. This framework targets ATLAS HLT and Offline computing and is designed for achieving parallelism in a heterogeneous environment potentially consisting of CPUs and massively parallel coprocessors, such as GPUs or Xeon Phi. The system also aims to provide automatic memory management and task scheduling over multiple coprocessors [6] [7].

Like the first design, APE also places GPU code in a separate process. However, instead of exposing GPU kernels to Athena directly, the APE server hosts software objects called *modules*. Each detector implements each own module to handle offload requests. A module contains the code for producing and completing *work items*; it is also responsible for event batching and management of any state kept between executions of different *work items*, such as: GPU constants, variable data, and streams. Different work items correspond to different algorithms, and a work item is usually composed of GPU kernels. The benefit of this design is that the work items could potentially be managed at a higher level: the server could be programmed to distribute CUDA work items over multiple accelerator cards based on their memory requirements. It may also be possible to share resources when executing multiple work items; however, this is complicated in practice.

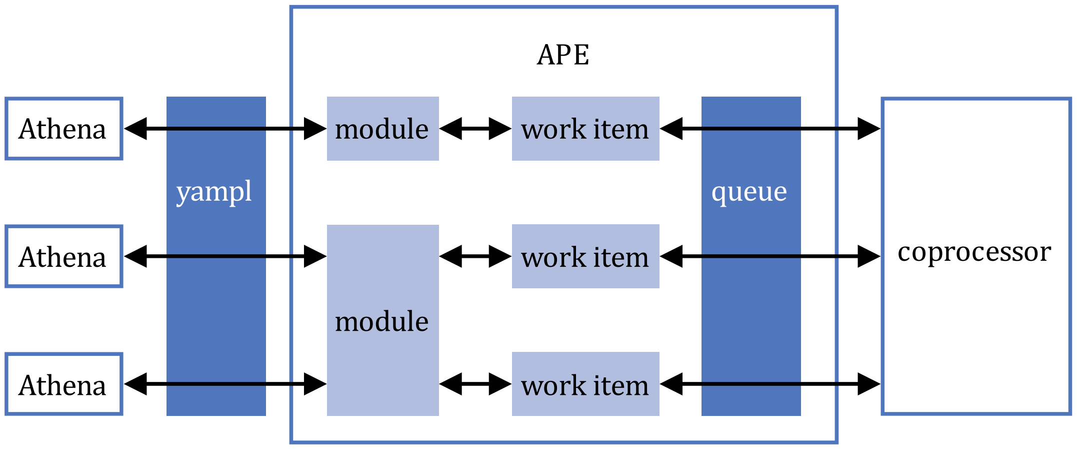

Figure 15 ATLAS APE architecture.

Rather than using custom communication protocols, ATLAS's APE relies on *yampl* (**y**et **a**nother **m**essage-**p**assing **l**ibrary) [67] for passing information between Athena and the APE server. *Yampl* hides the communication mechanism and protocol from the algorithm writer and allows for passing simple data types and arrays of simple data types between local processes or across a network.

Figure 16 compares APE's performance on three different parallelized Kalman Filter algorithm implementations using 1, 2, or 8 clients to stand-alone non-ATHENA versions using the CPU, OpenCL, OpenMP, and CUDA. The GPU versions use 5x5 GPU threads per track, and the OpenMP version uses 8 CPU threads. The test system uses an Intel Xeon E5 1620 CPU with 8 GB RAM and an NVIDIA GeForce GTX Titan GPU with 6 GB RAM. A data set consists of 96 events containing 19,500 tracks and 220,000 hits.

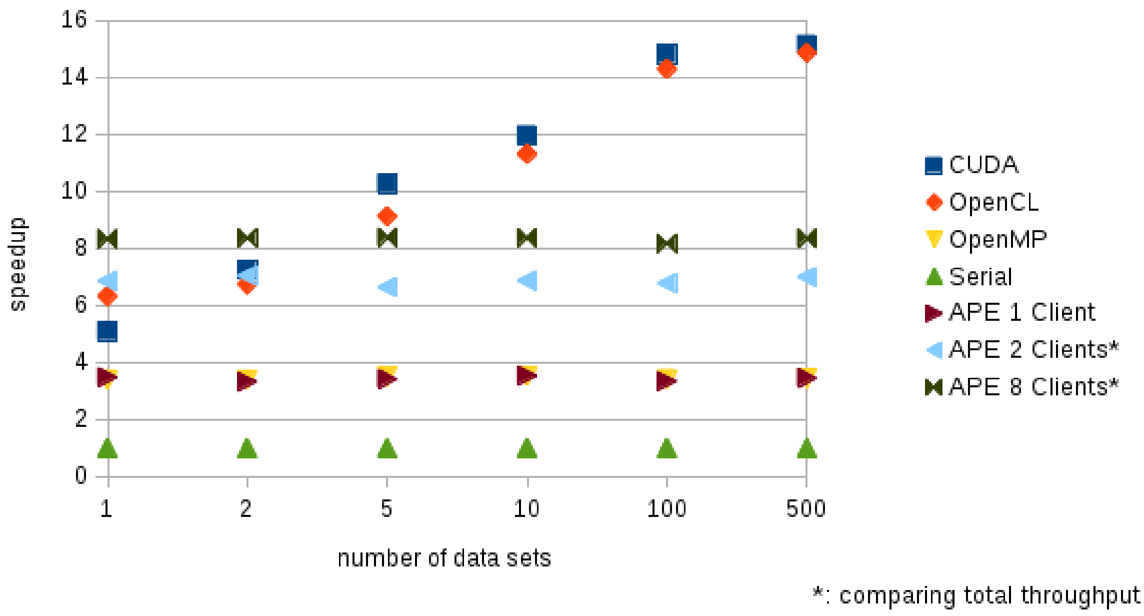

Figure 16 APE performance compared to standalone versions **[66]**.

The figure shows that standalone GPU implementations provide a progressively larger speedup with the number of data sets increasing up to 100, whereas the speedup provided by APE remains constant for a given number of clients.

6 Towards a new system

There are a number of issues to consider in the design of a system for massively parallel computation offloading integrated with the Gaudi framework:

- how to organize access to coprocessors;
- how to communicate between multiple running processes;
- how to exchange data for computations;
- and how to make the system available to Gaudi components.

Each choice comes with different costs and benefits. This chapter builds on the overview of the LHCb infrastructure given in previous chapters and the knowledge gained from examining the state of the art to survey the space of possibilities.

6.1 Coprocessor access

The main question is whether coprocessor access even has to be managed in any way. The simplest way to proceed would be for each algorithm to select whatever tools its author is most familiar with and access the coprocessor directly. There are several arguments to be made for creation of an intermediary and several different ways in which it could be implemented.

The GPU computation model makes it difficult for multiple processes to efficiently access the hardware at the same time. GPUs are typically designed for massively parallel computations of a single kernel or a small number of kernels by a single application (identified by an application context ID). The hardware is gradually evolving towards greater generality, but it is still far from CPUs in this aspect. Even in the future, however, massively parallel computation derives its efficiency from executing many similar computations on multiple pieces of data, so we want to concentrate their resources on few users at a time. Since typically multiple Gaudi instances run on a single machine, free-for-all GPU access could cause contention between the different instances for the limited GPU resource and create a bottleneck.

Another difficulty is the high setup cost that the user must pay to begin using a GPU. In our experiments, we observed a setup cost of 0.4 seconds — this is orders of magnitude greater than typical event processing time. Some management system would be required to avoid paying this cost for every GPU algorithm invocation. A GPU CUDA service could be provided, for example, to set up the device only once per Gaudi process; alternatively, all GPU algorithms could be confined to a single process exchanging data with the others, so that setup is performed only once in total.

Another issue to consider is whether a single event provides enough data to fully utilize the coprocessor. ALICE and NA62 experiments observed that a single event may not be enough. This is especially relevant for LHCb, which has small events compared to the other experiments. If data from multiple events is to be processed concurrently by a single algorithm, there has to be some means of gathering data from multiple events and submitting it to the algorithm in batches. This could also be facilitated by Gaudi Hive, introduced in section 3.5.

If there must be a coprocessor manager that spans multiple processes, the question of where to place its functionality arises. There should be at most one process responsible for each coprocessor device. One option is to designate one of the Gaudi processes as the server, another is to spawn a separate process for the coprocessor manager.

Finally, there is the question of how to organize access to the coprocessor within the coprocessor manager's process. A thin management layer could limit itself to organizing the data and handing it out to coprocessor algorithms. The advantage of this is to simplifying algorithm migration from standalone versions and allowing greater diversity of implementations. A thick layer, as used by ATLAS APE, reviewed in section 5.5, could force algorithms to go through an intermediate layer before accessing the GPU, but potentially enable more optimal use of resources.

6.2 Interprocess communication

A typical event farm node can filter and reconstruct many events concurrently, which requires spawning a Gaudi process for each event. Gaudi Hive adds the capability of handling multiple events in a single process, but it is currently still a work in progress. A coprocessor manager that allows event batching and minimizes setup overhead would have multiple Gaudi processes working together. It can also be useful to have the ability to exchange data with sources other than Gaudi for the purposes of generality, debugging, and testing.

LHCb's event computer farm runs on SLC (**S**cientific **L**inux **C**ERN), which is an operating system based on Red Hat Enterprise Linux. Like most operating systems in common use, SLC assigns a virtual address space to every process, meaning that every process sees only its own data unless it explicitly shares a region of memory with another. In order to gather event data from multiple Gaudi processes in a single algorithm, the processes have to communicate with each other; this type of communication is called IPC (**i**nter**p**rocess **c**ommunication). In the following text, the process running the algorithm and accessing the coprocessor is called the *server*, and the processes that send their data to the algorithm, the *clients*.

IPC requires the means of data transfer and synchronization between processes; SLC offers a range of options for this. For IPC to work, there also has to be a functional interface between the client and the server — a convention for how to invoke certain functions and how to send and interpret data. Communication is usually implemented in a separate layer, so that the other components have minimal knowledge of its internals. Yampl and Thrift libraries exemplify two common approaches to the communication layer's implementation.

ATLAS APE, described in section 5.5, automates IPC using the yampl library. This library allows passing simple "trivially copiable" data structures that are laid out contiguously in memory between processes. It is also necessary that the data structures are laid out in exactly the same way in the sender and the recipient, guaranteeing which requires the developer to take some special care. The library provides a range of communication methods for network communication, large local messages, and small low-latency local messages. The objects shared by yampl are just data; the library does not attempt to address the problem of implementing the functional interface between processes.

Two other common choices for IPC libraries are ZeroMQ [68] and Apache Thrift [69]. ZeroMQ was developed as a solution for distributed software. It provides a high-performance message transport layer that could be used within a single process, between processes on a single machine, or over a network. It provides a simple interface similar to sockets and some additional functionality for client and servers-side queueing and connection recovery.

Apache Thrift takes a more comprehensive approach to IPC. It was developed in 2007 at Facebook by a former Google engineer. Thrift allows transmission of more complicated datatypes and adds support for versioning and compression. The advanced type support is accomplished via code generation: a special compiler tool takes type descriptions supplied by the user and generates the code for transmission of objects of these types. Versioning provides for the possibility of evolving data types on the server side without breaking client code, and a fast compression scheme reduces the size of data in transit at the cost of a small CPU overhead.

Most importantly, Thrift allows augmenting data types with functions that act on the objects being transferred. A function can be implemented on the server and called on the client or vice versa, with all the input and output transferred transparently to the user — what is called RPC (**r**emote **p**rocedure **c**all). This makes it possible to define interfaces between processes. Like yampl, Thrift provides a range of communication methods.

There are two types of users whose concerns determine the choice of communication infrastructure: the coprocessor manager's developer and the algorithm developer. Depending on the coprocessor manager's design, the choice may affect them to different extends. For example, the algorithm developer may be given the means of communication between processes, or only a say in how a particular piece of data is transferred, or the manager may do it all on its own.

6.3 Data management

The communication problem is linked to the problem of data management. Data often comes in complicated structures comprised of interlinked pieces spread over separate memory locations. When the algorithm processing the data is located in the same process, it can access all of the data by following the references, but an algorithm residing in a different process needs the data to be copied and all the links updated to point to the new memory locations of all the pieces. The process of packing data into a contiguous block with no references to external data for transmission and the process of its restoration on the other end are called *serialization* and *deserialization*. As with IPC, there has to be a convention for the interpretation of the data between serialization and deserialization, or else the data may be corrupted. Maintaining this convention manually is prone to error, so special tools exist for automating this task.

The two most popular serialization tools are Google Protocol Buffers [70] and Apache Thrift, already introduced in section 6.2. Protocol Buffers were developed by Google in 2001 for lack of good existing solutions and released to the public in 2008. They are used by most of Google's services and are known to be very reliable and efficient. Both Protocol Buffers and Thrift use user-supplied interface definition files to generate serialization and deserialization code. Unlike Thrift, Protocol Buffers provide no data transfer or RPC support. This allows them to be mixed and matched with different solutions, but with the drawback of not making full use of the data type definition files if the solutions do not support them.

An alternative approach is to use data that can be transmitted as is, without any serialization. This is important, because serialization and deserialization of data can itself consume a significant portion of the total processing time. Yampl's "trivially copiable type" concept is the simplest use of this idea. Such data can be used normally by the algorithm, and then copied by yampl like a simple block of memory. It remains up to the developer to make sure that the type really ease trivially copiable and that it is encoded exactly the same way in all the processes that use it.

Cap'n Proto [71], developed by Protocol Buffers' primary author, uses a more sophisticated variation of the trivially copiable type concept. Cap'n Proto defines a data format that can be transmitted without any additional encoding, but allows references between objects, as well as variable-length data members like lists and text strings. The trade-off is that the data is not accessed directly, but rather through accessor functions generated from type definitions similar to Thrift's and Protocol Buffers'. These accessors are highly efficient and require much less generated code than Protocol Buffers. Cap't Proto also provides tools for communication and a high-performance RPC protocol that allows chaining remote calls without needlessly passing intermediate data objects to the client.

6.4 Gaudi integration

The Gaudi integration requirement must be taken into account in the coprocessor manager's design. There has to be a convenient method of interaction between Gaudi components and the coprocessor manager. If the manager requires any configuration, it should be configurable from standard Gaudi scripts. If it requires additional build-time tools, such as code generators, these have to be made available through LHCb's build system, described in section 0.

The most straightforward way to expose the manager's functionality is through a library. A library could be used by any application; it would contain the code for locating the coprocessor manager's server process, establishing a communication channel, and accessing its interface. The library would have to rely on the components using it for its configuration.

Another way is to expose the manager's functionality through a Gaudi service. In this case Gaudi algorithms would have a simple standard way of accessing it. The service would be easily configurable through Gaudi scripts and, if needed, be able to handle component state transitions, explained in section 3.2. This is a more idiomatic way of providing services within Gaudi.

If the design requires additional build-time tools, these have to be easily configurable through the build system. These tools have to be available to all the projects using the coprocessor manager, and there should also be a way to attain and configure them for users outside the LHCb network. If the tools are updated in a way that is not fully backwards compatible, there should be a way of choosing the version and migrating from one to the other. All these considerations make introduction of additional tools undesirable in general unless there is a very strong case for them.

6.5 Apache Thrift experiment

While studying the different design options, we experimented with building a computation offload system based on Apache Thrift. Seeing the error-prone original design of the ATLAS offload system, we tried to automate as much of it as we could. Apache Thrift offered a way of organizing transparent and safe communication between processes for client-server communication and data exchange.

Thrift requires two things two work: a code generator and a library. Thrift's code generator is a tool that converts type definitions written in a custom Thrift language to source code that defines the types and the target programming language and contains the instructions for network communication. The bulk of network communication code that is not specific to the generated types is contained in the Thrift library. The code generator has to be available at the time the offload system is built. The library has to be available at run time.

At the time, the Thrift library was incompatible with LHCb's runtime environment; it depended on components that were not available on all the systems we needed to support. With Thrift being an open-source project, we implemented our own version of the C++ library, custom-tailored to the environment and our needs.

Thrift type definitions were used in two ways. First, we wrote the type definitions that made up the interface for the offload service's client and server communication interface; all the commands were sent using RPC via Thrift object methods. Second, algorithm authors would write type definitions for the data their algorithms needed to process. This meant that the code generator needed to be present when the offload system was built, as well as for building the algorithms. The algorithms would link to the offload system's library, which included Thrift communication components.

The requirement of adding a Thrift code generator to the environment in every setting the offload system or an algorithm need to be developed is a significant cost that needed to be justified by equal benefits. Additionally, learning the Thrift data type definition language and writing the type definitions was placing an additional burden on the algorithm developer.

In practice, from the algorithm developer's point of view, the RPC capabilities offered by Thrift were unneeded, and copying between Gaudi and Thrift objects added unnecessary overhead. At the same time, manual serialization usually was not difficult as long as the communication part was taken care of. In the end, we decided against using Apache Thrift in the computation offload service.

7 Coprocessor Manager

The preceding chapters set up the context for explaining the focus of this work — the computation offloading service called the Coprocessor Manager. They cover the hardware this service is designed to serve, the software environment in which it operates, the problems it has to solve, and the choices facing the designer of any prospective solution. The current chapter builds on this context to describe the Coprocessor Manager in full detail.

This chapter begins by introducing our solution, defines its scope and usage. The next section explains the design decisions we made in the terms defined in Chapter 6. The last section covers the individual components that make up the system.

7.1 Overview

First, it is best to give a quick overview of the scope of our solution. The Coprocessor Manager addresses the problems that arise when trying to use massively parallel computation accelerator hardware directly from Gaudi algorithms, as covered in Chapters 5 and 6.

The Coprocessor Manager consists of several components; the main component is a computation server application — a host for algorithms that use massively parallel accelerators, such as GPUs. There could be a server application running on every node, or several nodes could share an instance and communicate with it via a network. As multiple Gaudi clients send data to it, the server application batches requests to algorithms and schedules them to maintain high throughput.

The other components facilitate integration with Gaudi, so that existing pipelines can make use of the new algorithms. It also includes tools that aid algorithm development within LHCb's software environment.

Here is, briefly, the Coprocessor Manager's main use case — offloading computations from Gaudi pipelines:

- the server application is started before any Gaudi pipelines are executed;
- the server application is instructed to load any required coprocessor algorithms;
- Gaudi pipelines are configured with the Gaudi algorithms that talk to the server application;
- the Gaudi pipelines run for as long as necessary;
- the server application is shut down afterwards.

The next section describes the design in detail. Instructions for using the individual components come last.

7.2 Design

There could be many different offload manager implementations that address the set of posed problems. All the different implementations can be divided into classes based on the several major choices defined in Chapter 6. The following section describes the choices we made with our implementation and our reasons for making them.

7.2.1 Coprocessor access

We first have to decide which process will access coprocessor hardware. Typically, a node will run multiple processes, each taking an event through a Gaudi pipeline. Only a small portion of events are fully reconstructed; most pipelines terminate early when they conclude that the event is not necessary for any ongoing physics experiment. Following the arguments made in section 6.1, we want to manage access from a single process and minimize overhead by having it run until the last pipeline terminates. We could devote one of the Gaudi processes for this task, but it is more natural to have a separate process just for the Coprocessor Manager.

There are a number of advantages to having the Coprocessor Manager run in a separate process: it does not interfere with pipeline management, it can be configured optimally for its task, and it can potentially be restarted without loss of data in case of a crash. It also reduces its dependency on Gaudi, so that it could be used independently for testing and profiling.

The next choice is about the thickness of the management layer. The Coprocessor Manager could restrict algorithms in various ways to extract more efficiency savings. It could manage memory or schedule algorithms across multiple coprocessors. In our experience, most algorithms are developed independently of the Coprocessor Manager and it is important to make the process of transferring them under its umbrella as painless as possible.

In our design, the Coprocessor Manager acts as a host for algorithms that require coprocessor access. The Coprocessor Manager decides when an algorithm runs and what data it processes, but hosted algorithms are given unrestricted exclusive access for the duration of their execution and can use any libraries for working with the coprocessor their authors prefer.

In order to be picked up by the Coprocessor Manager, a hosted algorithm must be implemented as a Gaudi project that depends on the *CpManager/ICpAlgorithm* project and generate a component library implementing a simple interface, called *ICpAlgorithm*. The interface provides an entry point to the algorithm and a way to provide it with input data and collect its output. It should be easy to adapt most independently developed algorithms to this interface.

7.2.2 Interprocess communication

Section 6.5 describes our experience with integrating the Coprocessor Manager with Apache Thrift. After we decided that it was an excessive burden compared to the advantages it provided, we went with a lighter option. Interprocess communication in the Coprocessor Manager is handled by small custom library, called *CpIpc*. The library transfers data of a number of basic types across the network. The transfer method can be chosen by the communicating processes, and new methods can be added. We currently have communication via Unix Sockets for situations, when the Coprocessor Manager runs on the same machine as all the Gaudi pipelines that talk to it. We also have TCP/IP communication for communicating over a network. Shared memory and Infiniband communication could be added in future.

The Coprocessor Manager assigns a dedicated server process to every machine equipped with massively parallel accelerator hardware. Data is sent from client processes that could be Gaudi or *cpdriver* instances. *CpIpc* handles the communication.

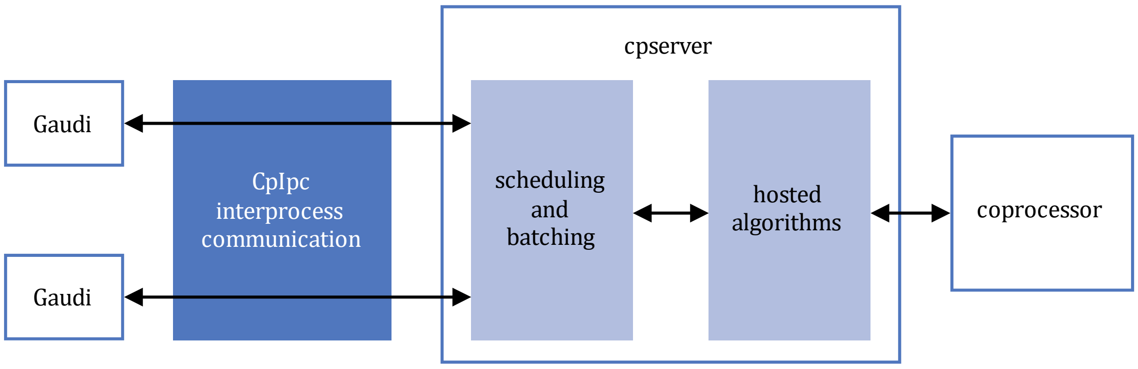

Figure 17. Coprocessor Manager architecture.

The server receives data from multiple concurrent clients, schedules algorithm execution, combines data into batches, runs algorithms, and distributes the results back to clients. The server does not enforce any constraints on the algorithms' use of the hardware resources, so algorithms that have been written without the Coprocessor Manager can be ported with minimal changes. However, the server could be augmented with additional hardware management capabilities, should they be deemed beneficial.

To handle multiple concurrent clients, the server opens a socket and accepts each connection on a new thread. This is currently a local Unix socket, but an option to use a network socket could be added with only small modifications. When a client connects, the server receives a data payload and the name of the algorithm to be used to process it. The payload is placed into a queue for scheduling; meanwhile, the thread is suspended. Once the payload is processed, the client thread is woken up, and the result of the computation is sent back to the client. The client gets notified when the requested algorithm is not available or an exception is thrown during the computation.

Algorithm scheduling and execution run on a dedicated thread. This has the benefit that the coprocessor is not stalled while data is passed back and forth between the server and its clients. Algorithms are executed one after another whenever data is available. The scheduling algorithm is a variant of the first-come-first-served FCFS approach: the difference is that whenever a payload is removed from the queue, all payloads targeting the same algorithm are removed from the queue with it. These payloads are submitted to the algorithm as a batch. When the hardware cannot cope with the amount of incoming data, new clients are refused connection.

FCFS, despite its simplicity, has an important benefit: because there is no prioritization, every payload is guaranteed to be processed in time. Prioritization schemes risk “starving” low-priority clients. However, more complicated approaches may be preferable, if latency guarantees are desired.

7.2.3 Data management

The *CpIpc* library provides the means for sending simple types between processes. This helps to define interprocess communication interfaces in a simple and reliable manner. However, the data types required for algorithms are often not so simple. When we were experimenting with Apache Thrift, we saw that serializing data for IPC is often too simple to bother with Thrift’s type specifications. The automation also involved sometimes unnecessary copying, which significantly impacted performance. The Coprocessor Manager chooses to give the algorithm developer control over serialization and copying. Serialization is introduced in section 6.3.

The data to be sent from Gaudi to the Coprocessor Manager is taken as a single memory block. If the algorithm uses a data structure that could be copied without modification, like the data types accepted by the ATLAS APE’s yampl library described in section 5.5, then the developer can pass the object, and no additional copies will be made. If the algorithm requires a more elaborate data structure, the algorithm developer could write serialization and deserialization code himself or use a tool like Protocol Buffers or Cap’n Proto.

Gaudi processes send data to the Coprocessor Manager using the *ICpService:: submitData* function call. This call takes a block of input data, waits for it to be processed, and then returns the output data. The way we return the output requires special consideration. Some interprocess communication methods receive data in blocks. We could have allocated a temporary memory buffer and copied the received data there, but this would have created a sometimes unnecessary memory copy. Instead we ask the *submitData* caller for an allocator function, which lets the algorithm developer provide a buffer of the requested size when needed. This way, a Gaudi process could avoid allocating memory for every event and keep a cache, if needed.

7.2.4 Gaudi integration

The Coprocessor Manager hosts coprocessor algorithms and manages multiple concurrent requests to process data using these algorithms. The requests could come from Gaudi event-processing pipelines or from some other sources; it could be a different framework or the *cpdriver* tool we have built to "play back" previously recorded sessions for the purposes of debugging and regression testing. While the Coprocessor Manager keeps its options open, it takes additional steps to integrate with what is currently its primary environment, Gaudi, to simplify algorithm development and deployment. Fitting in with established conventions is also important for software maintenance.

The Coprocessor Manager extends the Gaudi framework and uses its special features in two ways. The first extension is the addition of the *CpService* Gaudi service component. *CpService* simplifies data exchange between GPU algorithms and existing Gaudi components. The second extension uses the Gaudi plugin service to allow GPU algorithms to be developed within Gaudi alongside the existing software and be picked up by the Coprocessor Manager server automatically.

We chose to develop a Gaudi service to simplify access to the Coprocessor Manager for Gaudi components instead of a simple library because it allows access to be configurable in a consistent way and similarly to other Gaudi components. Some configuration is necessary to communicate with a Coprocessor Manager server. The server is typically identified by a local pipeline path or a network TCP host/port pair, depending on the type of connection for which it is configured.

The other point of integration into the Gaudi framework consists of allowing coprocessor algorithms to be developed alongside the rest of Gaudi components and added without the need to modify the server. Gaudi offers a plugin service, which loads components implementing specially marked interfaces by name. The Coprocessor Manager requires that GPU algorithms implement the *ICpHandler* interface; it uses the plugin service to load these algorithms at run time. All the algorithms implementing the interface and built on the node hosting the Coprocessor Manager are available automatically; there are no additional tasks the developer needs to perform.

It is important to note that, while a plugin service is necessary for the Coprocessor Manager, and currently Gaudi's existing facilities are used for this purpose, a different plugin service could be adapted if needed. Plugin service integration is different from the *CpService* component, which provides a convenience for Gaudi users without making Gaudi a requirement. This is not a fundamental limitation, but simply adherence to the environment's conventions. If the Coprocessor Manager were to be used with another framework, the server could be adapted to use that framework's plugin service or to use whatever plugin scheme is appropriate for that environment. Then the Gaudi requirement could be dropped.

7.3 Gaudi Hive considerations

The Coprocessor Manager tries to process data in batches, so the more concurrent requests it receives, the more efficiently it can schedule their execution. Gaudi Hive could become a more efficient way of sending many concurrent requests to the Coprocessor Manager. The implementation remain the same as with regular Gaudi: a Gaudi algorithm stub that sends data to the Coprocessor Manager and waits for the results, but this algorithm stub would be executed by the Gaudi scheduler as a *tbb::task* instance.

However, having two schedulers, one in *cpserver* and one in Gaudi Hive, creates an additional difficulty: a task the waiting for the *ICpService::submitData* call to return would stall Gaudi Hive's execution. Gaudi Hive uses Intel TBB for thread and task management, and TBB enforces a non-preemtive task scheduling strategy designed for CPU-bound computations — it has no concept of idle waiting.

In order to cooperate with the Coprocessor Manager, Gaudi Hive requires a way for tasks to suspend themselves or make better use of CPU multithreading to execute multiple tasks in parallel. This capability is also required for tasks performing disk or network IO. There has been some progress in this area [72], and it is possible that this capability will be added in future.

If coprocessor algorithm performance is significantly degraded by the CPU scheduler, Amdahl's Law, discussed in section 4.2, can make any throughput gains moot [73].

7.4 Components

7.4.1 cpserver

The Coprocessor Manager includes two important executables: *cpserver* and *cpdriver*. *Cpserver* implements the service that hosts algorithms and communicates with Gaudi or other data sources. *Cpdriver* is a utility that replays pre-recorded data to *cpserver* and verifies the results; it is also an example of sending data to *cpserver* from a source other than Gaudi.

Cpserver has two common use cases: launching a new instance of the server process, and controlling one that is already running. *Cpserver* takes the following arguments:

- **daemonize** – runs *cpserver* as a daemon. Without this argument, it runs as a regular console process.
- **exit** – terminates a previously launched instance. The instance is identified by the *service* argument.
- **load** – loads an algorithm into a running instance by name. For the *PrPixelCuda* algorithm, for example, the name is *PrPixelCudaAlgorithm*.
- **connection** – *local* or *tcp*; toggles the means of communication with clients.
- **localPath** – a local Unix socket path, used for *connection* set to *local*. This is a high-performance means of communication within a single machine.

- **tcpHost** – network socket address, used for *connection* set to *tcp*.
- **tcpPort** – network socket port, used for *connection* set to *tcp*.
- **datadir** – enables recording of input and output data to disk. Specifies the directory, in which to store the files. The directory should exist.

Different connection types use different parameters to identify the server. When *connection* is set to *local*, the server is identified by a local Unix socket path, which should be a valid file name. When the *connection* is set to *tcp*, the server is identified by host and port.

The server additionally keeps a log of algorithm performance. The log is saved to the file *perf.log*. Every time an algorithm processes a data batch, a new entry is added to the log, listing a time stamp, the name of the algorithm, the time it took to process the batch, the number of entries, the size of the input, and the size of the output.

When *datadir* is specified, each client connection, such as one made by a *submitData* call, generates a file containing the data received and the data produced by the algorithm used to process it.

7.4.2 cpdriver

The data files recorded by *cpserver* can be played back using the *cpdriver* tool. *Cpdriver* takes the following arguments:

- **connection** – *local* or *tcp*; toggles the means of communication with *cpserver*.
- **localPath** – as with *cpserver*, a local Unix socket path.
- **tcpHost** – network socket address. Used for communication across a network.
- **tcpPort** – network socket port.
- **threads** – the number of threads used to send the data.
- **verify** – have *cpdriver* check the output returned by *cpserver* to the output recorded originally and produce a message in case of mismatch.
- **data** – path to the directory containing record files. The file naming convention should be the same as used by *cpserver* with the *datadir* argument.

Playing back recorded data can be used to measure hosted algorithm performance without data production algorithms interfering. It is also useful for maintaining algorithm correctness: running a previously recorded data set with the *verify* flag after modifying an algorithm is a form of regression testing to make sure that the algorithm still produces the same output.

As described in section 0, LHCb's code based is organized into large projects made up of packages. One package can depend on others, but as long as its dependencies are satisfied, it can be moved between projects. Here is sample Coprocessor Manager use with event reconstruction in Brunel.

```
# configure the server to copy incoming data to the MyEvents directory
> cpserver --datadir MyEvents &
> cpserver --load PrPixelCudaHandler
  loaded handler 'PrPixelCudaHandler'

# get events from Brunel and send them to the Coprocessor Manager
> gaudirun.py Brunel-Script.py
  ==============================================
            Welcome to Brunel version v45r0
   running on lab13 on Fri Dec 31 23:59:59 1999
  ==============================================
  …

# replay the events one more time, checking whether the results are
consistent
> cpdriver --data MyEvents --verify
```

Listing 1 Sample usage of *cpserver* and *cpdriver*.

7.4.3 CpIpc

The communication protocol and network machinery are encapsulated in the *CpIpc* library. The library provides the framework for establishing client-server connections and safely exchange data of several basic datatypes: boolean, double-precision floating point, unsigned 32-bit integer, string, and binary data. The transport layer is extensible with different means of data transfer, such as local Unix sockets and TCP sockets. The architecture is broadly similar to Apache Thrift.

The server uses this library to communicate with its clients and with other instances of itself, such as when the *cpserver* executable is used to terminate a running previous instance or load an algorithm into it. *CpService* uses this library to allow Gaudi algorithms to submit data to *cpserver*. The *cpdriver* executable uses this library to send pre-recorded data to *cpserver*. Potentially, this library could be used to enable additional types of *cpserver* clients or even different kinds of servers for existing clients.

7.4.4 CpService

As explained in section 3.2, algorithms running on Gaudi have access to various services. The Coprocessor Manager exposes a service of its own via the *ICpService* interface, which is instantiated in the standard way for Gaudi services:

```
ICpService cpService = svc<ICpService>("CpService", true);
```

Listing 2 *ICpService* initialization.

ICpService exposes only a single function: *submitData*. This function takes a data array and the name of the hosted algorithm to be used, sends the data to *cpserver*, waits for a response, and returns another data array that is the result of running the algorithm. If the client can perform other useful work while waiting for results, *submitData* can be safely called from a worker thread. Here is its signature:

```
class ICpService : public virtual IInterface {

  virtual void submitData(
    std::string  algorithmName,
    const void * data,
    const size_t size,
    Alloc        allocResults,
    AllocParam   allocResultsParam);
};
```

Listing 3 *ICpService* definition.

There are two parameters that need to be explained: *allocResults* and *allocResultsParam*. These parameters form a pattern that allows the caller of *submitData* to choose its own memory allocation method for reception of the result data, thus potentially avoiding creation of unnecessary intermediate copies. The former parameter is a user-supplied function of type *Alloc* that takes the amount of memory to allocate and returns a corresponding memory block. The latter parameter is an arbitrary user-supplied parameter that will be passed to *allocResults* along with the memory amount.

```
/// User-defined data to be passed through to the allocator.
typedef void * AllocParam;

/// Allocator function.
typedef uint8_t * (*Alloc)(size_t size, AllocParam param);
```

Listing 4 Allocator definition.

A simple implementation could pass a *std::vector* as *AllocParam* and resize it in *Alloc*. *Alloc* would then return a pointer to the contents of the *std::vector*.

```
uint8_t * allocVector(size_t size, void * param)
{
  typedef std::vector<uint8_t> Data;
  Data & tracks = *reinterpret_cast<Data*>(param);
  tracks.resize(size);
  return tracks.data();
}
```

Listing 5 Sample allocator implementation.

CpService declares four configuration parameters for locating the *cpserver* instance to which it will send the data. These parameters should be set in the pipeline's Gaudi script.

- **ConnectionType** – *local* or *tcp*; toggles the means of communication with *cpserver*.
- **SocketPath** – a local Unix socket path, used for *ConnectionType* set to local.
- **TcpHost** – network socket address, used for *ConnectionType* set to *tcp*.
- **TcpPort** – network socket port, used for *ConnectionType* set to *tcp*.

7.4.5 CpAlgorithm

To be made available for loading via *cpserver*, an algorithm needs to add a dependency on the *CpManager/CpAlgorithm* project and generate a component library implementing the *ICpAlgorithm* interface. The *ICpAlgorithm* interface contains only a single function that the GPU algorithm must implement.

```
class ICpAlgorithm
{
  typedef std::vector<uint8_t>     Data;
  typedef std::vector<const Data*> Batch;

  virtual void operator() (
    const Batch & batch,
    Alloc         allocResult,
    AllocParam    allocResultParam);
};
```

Listing 6 *ICpAlgorithm* interface.

A key feature of the Coprocessor Manager is its ability to accumulate data from multiple Gaudi processes and hand them over to algorithms in batches. Hence, the algorithm receives an *std::vector* consisting of one or more data arrays sent by calls to *submitData*. Most simply, it could process these data arrays one by one, ignoring the batching capability. Alternatively, it could schedule multiple concurrent kernels to process multiple data arrays at a time or agglomerate them all into a single data block and process all in a single pass.

As with *ICpService*, the *Alloc/AllocParam* pattern allows the caller to supply a custom memory allocation function. However, this time these parameters are supplied by the Coprocessor Manager. The algorithm implementer is expected to call *allocResult* to request memory for the algorithm's results. The contents of this memory are eventually returned from *ICpService::submitData*. Again, this pattern allows us to avoid making unnecessary intermediate copies, while keeping a consistent interface for algorithm authors going forward.

The Gaudi framework requires one more addition. The **.cpp* implementation file for the handler needs to call the *DECLARE_COMPONENT* macro with the name of the implementation class. Listing 7 shows a sample implementation.

```
#include "GpuPixelSearchByTriplet.h"
#include "PrPixelCudaAlgorithm.h"

#include <algorithm>
#include <vector>

using namespace std;

DECLARE_COMPONENT(PrPixelCudaAlgorithm)

void PrPixelCudaAlgorithm::operator() (
    const Batch & batch,
    Alloc         allocResult,
    AllocParam    allocResultParam) {
  // analyze the event
  vector<Data> trackCollection;
  gpuPixelSearchByTriplet(batch, trackCollection);

  // trackCollection now contains a set of tracks for every event
  // in the batch; distribute them to callers
  for (int i = 0, size = trackCollection.size(); i != size; ++i){
    uint8_t * buffer =
      allocResult(i, trackCollection[i].size(), allocResultParam);
    copy(trackCollection[i].begin(), trackCollection[i].end(), buffer);
  }
}
```

Listing 7 Sample PrPixelCudaAlgorithm implementation.

8 VELO Tracking on the GPU

Track reconstruction is the procedure of estimating the most likely particle trajectories given coordinates of hits at the various subdetector sensors. Knowing the path of a particle through a fixed magnetic field allows its charge and momentum to be determined. Because every detector has a specific set of custom sensors, electronics, and requirements, there is no standard track reconstruction algorithm or library — every experiment comes up with custom algorithms that match its goals and constraints.

The VELO Pixel subdetector was introduced Section 2.2.1. Particles created in the interaction region pass through VELO Pixel before reaching the dipole magnet, so they travel in straight lines. This simplifies the task of finding their trajectories and allows for smaller, easier to understand algorithms. For this reason, a GPU-based tracking algorithm for a future upgraded version of VELO was chosen to evaluate the Coprocessor Manager's performance.

This chapter describes the algorithms used for performance evaluation and gives the background to help explain their performance. The background includes a classification of track types encountered in the detector, an explanation of the process of track reconstruction, and the description of the efficiency measurement methodology. Finally, the existing CPU-based VELO Pixel tracking algorithm is described in detail, and then the new GPU-based version.

8.1 Track types in LHCb

As described in Section 2.2, the LHCb detector consists of multiple subdetectors arranged in a cone spreading out from the collision site. Particles formed in the collisions leave tracks that have different characteristics: some start at the collision site, others some distance away from it; some tracks pass through all of the subdetectors, others fly out of bounds. Different types of tracks are treated differently by reconstruction algorithms. Figure 18 illustrates the different track types.

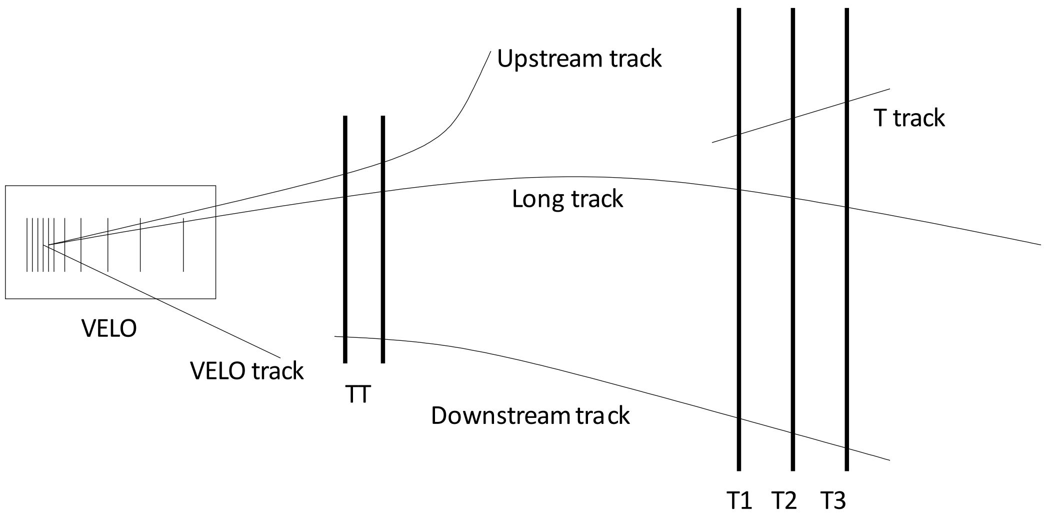

Figure 18 Schematic view of the different types of tracks in the LHCb detector **[74]**.

Long tracks traverse the entire tracking system, from VELO to the last tracking station, leaving hits in all the subdetectors. The tracks allow the most precise measurements of momentum.

Upstream tracks traverse only VELO and the TT stations. These low-momentum tracks are bent by the magnetic field out of the detector's acceptance before reaching the T1-T3 tracking stations. These tracks usually have poor momentum resolution.

Downstream tracks originate outside of VELO and have hits only in the TT and T1-T3 tracking stations. These tracks are usually produced by charged long-lived particle decays, such as those containing "strange" quarks.

VELO tracks have hits in the VELO station but cannot be matched to hits in the other subdetectors. They are useful for primary interaction vertex reconstruction.

T tracks are only observed in the T1-T3 tracking stations. These sometimes come from decays of long-lived particles and can be used in RICH reconstruction and for internal T station alignment.

8.2 Track reconstruction

When a particle passes through a silicon sensor, the VeloPix ASIC chip it is mounted on reports which pixels were hit; pixel (x, y, z) coordinates are known [31]. The chips provide no additional information for determining which hit corresponds to which particle. Track reconstruction's job is to match hits to particles and determine their most likely trajectories. It takes the hit coordinates from the detector and for each module computes track state vectors of the form:

$$TrackState(z_0) = (x_0 \quad y_0 \quad t_x \quad t_y \quad q/p)$$

Parameters x_0 and y_0 indicated position of the particle in the reconstructed track at the module's z-position z_0. Parameters $t_x = \partial x/\partial z$ and $t_y = \partial y/\partial z$ are the track slopes in the xz and yz planes. Together, these two variables indicate the direction of the particle's movement. The last parameter q/p is the estimated charge divided by momentum for the reconstructed particle. However, VELO tracking cannot measure this parameter because particles travel through VELO before being deflected by the dipole magnets.

Track reconstruction accuracy is improved by taking into account the physical properties of particles and fitting[1] hit coordinates to statistical estimates of likely tracks. One way to do this would be to treat the deviation of each point from the track as an error and minimize the sum of their squares. This is called least-squares estimation. This method has a unique solution, but it requires inversion of a large matrix and makes it difficult to account for such physical phenomena as multiple scattering and energy loss.

The Kalman filter [75] does not suffer from these drawbacks, making it especially well-suited for track reconstruction. Being an incremental algorithm, it also makes it possible to impose tolerances on tracks and quickly reject those that fail to meet them. Kalman filtering is used not only for VELO tracking, but also for full track reconstruction. In the general case, the Kalman filter consists of the following three distinct steps [76]:

- **Prediction.** Predict each node's state based on the previous nodes.
- **Filtering.** The prediction is updated according to the current node's measurement. The prediction and filtering steps are repeated for all measurements.
- **Smoothing.** Once all the measurements are added to the track, the track states are updated in the reverse direction, so that all measurements are properly accounted in every node.

The Kalman filter for VELO tracking is based on minimizing the hit measurement χ^2, calculated as the sum of squared distances from hits to the track, and therefore approximates the least-squares estimate. Given previous track coordinates (x_0, y_0, z_0) and (x_1, y_1, z_1), the fitting function is defined as:

$$fit(x_2, y_2, z_2) = \frac{x_{err}^2 + y_{err}^2}{(z_2 - z_1)^2}$$

$$x_{err} = |\bar{x}_2 - x_2|, \ \bar{x}_2 = x_0 + (x_1 - x_0)(z_2 - z_0)/(z_1 - z_0)$$
$$y_{err} = |\bar{y}_2 - y_2|, \ \bar{y}_2 = y_0 + (y_1 - y_0)(z_2 - z_0)/(z_1 - z_0)$$

[1] Hit fitting should not be confused with track fitting, which refers to the estimation of a track's parameters once the hits have been determined.

That is, the fitness is the error squared between the observed coordinate (x_2, y_2) and the predicted coordinate $(\bar{x}_2, \bar{y}_2)$, scaled by the square of the z distance between the last two hits. The prediction assumes that the particle moves in a straight line. The track is reconstructed incrementally, going along the z axis. At each step, the hit with the best fitness is added to the track.

8.3 Tracking efficiency

Several methods for measuring tracking efficiency exist, with the simplest one being to run the algorithm over simulated data. In simulation, we know all the particles that traverse the detector and can compute an objective efficiency measure. However, it is a priori unknown how well simulation data agrees with real collision data. Another method, known as *tag and probe*, uses real collision data to check the tracking algorithm for one subdetector by comparing to the tracking output of other subdetectors. The name comes from the separation of a particle's decay product tracks into fully reconstructed *tag* tracks and partially reconstructed *probe* tracks. Both reconstructions estimate the parent particle's invariant mass, and the *probe* track is considered efficient when the estimates match. [77]

Tracking algorithm efficiency is measured in simulation as the ratio of the number of correctly reconstructed tracks to the number of reconstructible tracks known for the simulation. LHCb tracking framework uses the following definitions [31]:

- A particle is reconstructible as a VELO track if there are clusters associated to it on three or more modules.
- A particle is considered reconstructed if at least 70% of the measurements on a track are associated with it.
- A **ghost** track is a track that cannot be matched to any simulated particle. If more than one reconstructed track is matched to a single particle, the extra tracks are counted as **clones**.

8.4 CPU-based VELO Pixel tracking

The current VELO Pixel tracking algorithm, called *PrPixel* proceeds in three distinct steps. The *prepare* step retrieves hit coordinates and initializes structures that will be used later. The *searchByPair* step reconstructs tracks based on hit coordinates and fills in the data structures initialized in the *prepare* stage. The *storeTracks* step converts these data structures into the standard format for use by other algorithms.

The *searchByPair* step takes up 78% of the algorithm's total execution time [78]. It is an efficient sequential algorithm that uses heuristics based on known track properties to break out of loops early and avoid doing unnecessary work. This property saves time, but also presents difficulties for porting the algorithm to parallel architectures. The *searchByPair* step proceeds as shown in Listing 8.

```
for sensor$_0$ in sensors in reverse order
  sensor$_1$ = sensor$_0$ - 2
  for hit$_0$ in sensor$_0$.hits
    for hit$_1$ in sensor$_1$.hits
      checkHitsAreCompatible(hit$_0$, hit$_1$)
      checkNoneOfTheHitsAreAlreadyUsed(hit$_0$, hit$_1$)
      track = generateTrack(hit$_0$, hit$_1$)
      cutOnR2Beam(track)
      addHitsInSensors(track, previous_sensors, 0)
      if sensor$_0$.z > MAXZFORBEAMCUT
        addHitsInSensors(track, posterior_sensors, 2)
      removeBadHits(track, MAXCHI2HIT)
      checkTrackHasAtLeastThreeHits(track)
      addHitsInSensors(track, [sensor$_0$, sensor$_1$])
      if numberOfHits(track) == 3
        checkAllThreeSensorsAreDifferent(track)
        checkAllHitsAreUnused(track)
        checkAllHitsAreChi2Good(track)
      else
        checkNumberOfUnusedHitsIsAbovePercentage(track, 0.6)
      tracks.addTrack(track)
      if numberOfHits(track) > 3
        markHitsAsUsed(track)
        break
return tracks
```

Listing 8 *PrPixel*'s *searchByPixel* pseudo-code.

The algorithm iterates through the sensor modules from back to front. At least three hits from different sensor modules are needed to create a track. Two of the hits have to be in adjacent modules, and the third can be in one of the preceding five modules or the following three.

Once a hit is included in a track, it is marked as used and taken out of consideration for other tracks. That is, it processes the hits in a “greedy” way. Consequently, the algorithm’s output is deterministic, but dependent on the order in which the hits are processed — different permutations of hits can yield different tracks.

8.5 GPU-based VELO Pixel tracking

8.5.1 Overview

PrPixel is a good illustration of a case where an algorithm highly optimized for CPU computation would not perform well on a massively parallel processor.

All common GPU frameworks, including CUDA, operate on the concept of a kernel. A kernel is a small program. The GPU spawns multiple kernel instances, so each can process a different piece of the input data. Efficient execution demands that almost all the instances should execute the same instructions: conditions should pick the same branch and loops should run the same number of cycles.

PrPixel's heuristics and greedy hit processing make it poorly suited for direct porting to the GPU. When multiple instances process different hits, they follow different computation paths based on the heuristics. If the instances run independently, they could pick some of the same hits for different tracks. A massively parallel implementation must address both of these issues.

8.5.2 Algorithm description

We ran our tests on a reconstruction algorithm developed by Daniel Cámpora [79]. The name of the package is *PrPixelCuda*, and the algorithm name for loading via the Coprocessor Manager is *PrPixelCudaAlgorithm*. This algorithm follows a similar scheme to *PrPixel*, but is specially optimized for GPU computation.

The algorithm collects hit coordinates for events from Gaudi, sorts them along the x coordinate, hands them off to *CpService*, and then waits for the results. On the *cpserver* side, the algorithm's handler is given hit coordinates in batches from multiple events and for each event compiles lists of hits likely to form tracks. The handler is implemented in Nvidia CUDA 7.0.

The algorithm performs a preparation step and then builds tracks iteratively, progressing through the sensors in order, starting at the far end of the detector. During the preparation step, it goes through all the sensors once and for each hit finds runs of second-next-sensor hits not exceeding a predefined x slope threshold; this is efficient, because tracks are sorted by the x coordinate. Only candidates from these runs need to be considered when building a track.

The track-building step is based on the idea of track following. A track is followed until no hits consistent with its trajectory are found over a span more than three sensors long. As it goes from sensor to sensor, the algorithm considers whether any track currently followed is consistent with any of the hits two sensors away, and, if so, extends the track with the best-matching hit.

At last, the algorithm examines every hit registered at the current sensor that is not yet part of any track and goes over all the hit triplets starting with this hit and compatible hits (gathered in the preparation step) in the two sensors that follow, finding all consistent with beginnings of new tracks. If such triplets are found, the most fitting is added to the tracks being followed.

The algorithm processes multiple events simultaneously using fast shared memory for communication between threads working on each individual event. The work of fitting hits to tracks is split among 128 threads per event in the current implementation.

8.5.3 Complexity analysis

We have not timed the separate stages of the algorithm, because massively parallel computation makes it difficult to obtain this kind of information, but we can find the time complexity of each. We measure performance relative to the average number of hits per module, which we call n. This number is proportional to the number of tracks, related to beam intensity. Running time is also a function of the number of modules in the sensor, but we consider this number fixed. Hence, the average number of hits per module is also proportional to the total number of hits, which determines the input size.

The preparation step sorts the hits along the x coordinate. This uses the C++ standard library sort algorithm, which has the average time complexity $O(n\log(n))$, same as worst-case complexity in most common implementations. Computation of candidate hit runs looks at pairs of hits from successive modules, therefore its complexity is $O(n^2)$. Candidates are collected from a fixed-angle cone starting at each hit, so the average number of hits per run is proportional to n. Therefore, reducing matching hit search to these runs increases the algorithm's performance by a constant factor without reducing its time complexity.

The track-building step compares currently followed tracks to hits. The number of currently followed tracks is proportional to n, therefore the complexity of this step is $O(n^2)$. Hit triplet examination is a $O(n^3)$ operation.

In total, the complexity of the algorithm is $O(n\log(n) + n^2 + n^2 + n^3) = O(n^3)$.

9 Performance evaluation

The current chapter examines the Coprocessor Manager's performance. It addressed two questions:

- how much overhead is added by the additional layer compared to direct access to the coprocessor,
- and how processing events in batches affects performance.

Each of the questions motivates an experiment. The overhead is measured by running sending random data to the Coprocessor Manager and not doing any work. The batching performance is measured by running a GPU-based VELO Pixel tracking algorithm and comparing it to the CPU version.

9.1 Hardware setup

We ran all our tests on a machine equipped with an Nvidia GeForce GTX 680 GPU and two six-core Intel Xeon E5-2620 CPUs. GTX 680 is a mid-range consumer graphics card; current top offerings are two to three times faster. The Xeon is a mid-range server CPU.

9.2 Overhead measurement

9.2.1 Introduction

Compared to processing data directly in Gaudi, the Coprocessor Manager introduces some additional overhead. As explained in section 6.3, input data sent to the Coprocessor Manager and the output returned to the client process has to be serialized before transmission and then deserialized on the other end. Serialization may be complicated for some algorithms while others may use data formats that allow copying it as is. Data transmission, batching, and algorithm execution scheduling add some overhead on top of serialization.

These additional tasks are performed on the CPU. Ideally, the CPU would be serving new data to the Coprocessor Manager while previous requests are being processed. In the worst-case scenario, the CPU would stall and do no useful work while waiting for the GPU. Efforts on improving task scheduling in Gaudi are being undertaken [72].

9.2.2 Experiment

The overhead added by the Coprocessor Manager can be split into latency increase and throughput reduction. In order to measure it, we use a placeholder algorithm that performs no computation on the data it receives and returns only a single integer as a dummy result. We send this algorithm varying amounts of random bytes via *cpdriver*. Communication is set up via local Unix sockets — a feature of the Linux operating system that provides fast interprocess communication over shared memory.

We sent 1,000 data packets with uniformly random sizes from 0 to 500,000 bytes, covering the range of likely event sizes. The *cpdriver* tool was used to record running time from the moment a data request is sent to *cpserver* to the moment the result is returned from it. This measures all the overhead added by the Coprocessor manager as a function of input data size. Serialization costs were not measured, since they vary from one algorithm to another and are sometimes entirely avoidable.

For the baseline, we measured the time it takes to send data with likewise distributed sizes between two local Unix sockets.

9.2.3 Results

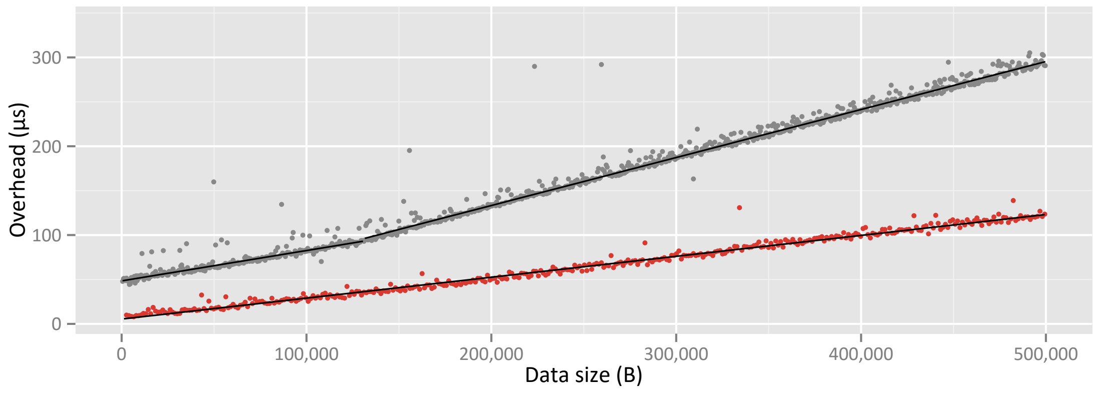

Figure 19. *cpserver* data transfer overhead in grey (above) and baseline transfer rate in red (below) along with the trend lines.

Figure 19 shows a simple linear relationship between data size and overhead time. Smaller overhead is better because it means that more time is spent on the algorithm and less on The Coprocessor Manager achieves a throughput of 2.7 GB/s for data packets under 128 KB and 1.7 GB/s for the larger. The base rate is 3.9 GB/s. The difference in rates for different sizes is not explicitly programmed into the Coprocessor Manager and may be hardware-dependent.

The minimal application does not use the GPU, so neither graph includes the overhead of GPU setup and memory transfer. We will see these costs when we look at batching performance in section 9.3.

The y-axis intercept for the *cpserver* data transfer overhead graph indicates the latency added by the server — it is the minimum amount of time before the client can expect to receive any response, even if no data is sent. We observe this delay to be 50 µs.

9.3 Event batching

9.3.1 Introduction

As explained in Chapter 6, we know from other experiments that batching data from multiple events is likely essential for full coprocessor utilization. The extent of the benefits will be different for different types and models of coprocessors and for different algorithms, but not so much that a case study would not be informative.

The VELO Pixel tracking algorithm introduced in Section 8.5 presents a problem that is computationally expensive, having $O(n^3)$ complexity relative to the average number of hits per module, and representative of a class of reconstruction algorithms. The approach it uses is optimized for GPUs, but similar to the current VELO Pixel CPU tracking algorithm, as well as to the earlier VELO tracking algorithm, FastVelo [80].

9.3.2 Experiment

The GPU-based tracking algorithm was used to process Monte Carlo simulated hit data from Brunel. The data was recorded to disk and then played back via the *cpdriver* tool using multiple concurrent threads. As in 9.2, the *cpdriver* tool was used to record running time from the moment a data request is sent to *cpserver* to the moment the result is returned from it.

The *cpserver* tool has been equipped with a modified scheduler that waits for a given number of requests to arrive before submitting the batch for processing. Each batch was processed by the track-forwarding VELO Pixel tracking algorithm. The waiting time is not included in timing.

The test was run with batches of 1 to 500 events in 1-event increments, making 10 runs for every batch size. The server was not restarted between runs. As we will see, this means that the first run of every set of 10 runs includes GPU initialization overhead.

The CPU baseline uses a September 2015 version of the highly optimized CPU-based VeloPix tracking algorithm *PrPixel* [31]. This is a sequential algorithm, running on a single logical core. We tested its performance using Gaudi and use the running time measurement that Gaudi provides.

It should be noted that, while *cpserver* allows batches to be as large as permitted by the operating system's limits on the number of simultaneous threads and open connections per process, our tests run up to the batch size limit imposed by the algorithm's implementation.

It is still uncertain whether keeping 40 events or more in flight per node would not cause scheduling problems for the Gaudi Hive framework. Current work looks at 20 events as a high number to manage [72]; much more could prove untenable. Gathering events from multiple nodes using a high-performance transport, such as *GPUDirect*, described in Section 4.4.3, could alleviate this difficulty, but more research is needed.

9.3.3 Results

Figure 20 summarizes the results for this experiment. The top graph shows that the first run for every batch size is delayed by 0.4 s. This is a one-time cost for GPU device initialization. Subsequent runs all complete quickly. The main benefit of massively parallel computation lies in the ability to increase batch size at almost no performance cost over wide ranges of batch sizes. The time per event falls rapidly as batch size increases: it is 2 ms for batches of 100 events, 1 ms for batches of 200, and 0.7 ms for 500. In practical use, we would have to weigh the increase in throughput attained by using larger batches against latency tolerances.

The rate of time increase over batch size ranges with no large jumps is also important. This slope shows the cost of processing greater of data amounts when the GPU has threads to spare and no additional processor time is spent. What is left is data encoding and decoding, as well as CPU-GPU memory transfers. The slope's gentleness indicates that this overhead is small relative to the computation cost. This is further elaborated in Section 10.1.

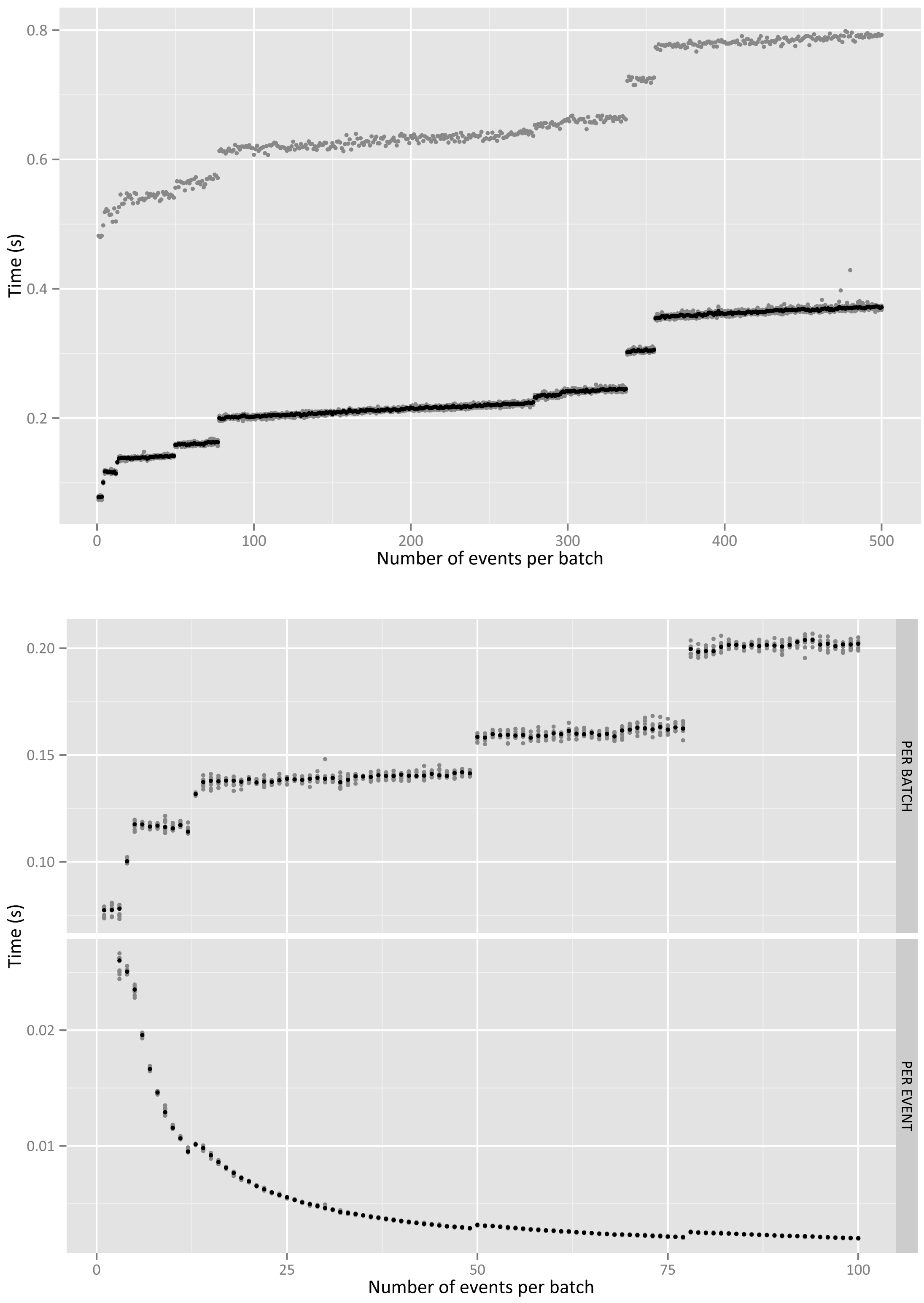

Figure 20. Times over 10 runs for VELO Pixel track forwarding running on *cpserver* with varying batch sizes: full dataset above and a zoomed in view below. Run data points are gray, median values black.

Table 3 makes the comparison between the CPU-based algorithm and the GPU-based algorithm in terms of time, ghost rate, and efficiency — as described in section 8.3, a ghost track is one that cannot be associated to any simulated particle, and efficiency refers to the proportion of simulated tracks detected. We see that the GPU algorithm pulls ahead of the CPU implementation for large batches, while lagging behind on single-event loads.

	PrPixel (CPU)	**Track forwarding (GPU)**	
Time per event (ms)	3.6	76.5	batch of 1
		26.2	batch of 3
		3.5	batch of 40
		2.0	batch of 100
		0.80	batch of 300
Ghost rate	1.7%	0.8%	
Efficiency for VELO tracks	95.6%	95.7%	
Efficiency for long tracks	98.3%	98.0%	
Efficiency for long B^0 decay tracks	99.0%	98.3%	
Efficiency for long strange tracks	97.7%	97.6%	
Efficiency for long tracks, $p > 5\text{GeV/c}$	98.8%	98.4%	
Efficiency for long strange tracks, $p > 5\text{GeV/c}$	98.3%	98.6%	

Table 3 Comparison of a CPU and a GPU VELO Pixel tracking algorithm.

10 Analysis

The previous chapter measured the coprocessor manager's overhead and the impact of batching events. We are now well-equipped to analyze the coprocessor manager's design's fitness according to the criteria established in the first chapter. That is, the new algorithms meet LHCb's performance requirements and that they deliver a significant benefit in terms of performance per dollar.

The current chapter splits the GPU-based VELO Pixel tracking algorithm's result from section 9.3 into throughput and latency components and compares them to the Coprocessor Manager's performance numbers from section 9.2, as well as LHCb's performance requirements. The second half of the chapter references tests performed with an OpenCL version of the same tracking algorithm on different types of hardware and gives the approximate throughput per dollar numbers.

10.1 Performance

We want to see whether the framework adds a significant overhead to algorithm execution and whether it succeeds in meeting LHCb Online's performance requirements. The first step is to analyze the GPU track forwarding algorithm's performance, measured in section 9.3, and separate it into throughput and latency.

Section 9.3 investigates a GPU VeloPix tracking algorithm's performance on batched events over a range of different batch sizes. The clear pattern is that performance increases in jumps, rising only gently in-between. The gently sloping runs of batch sizes occur where the GPU has threads to spare and adding more data incurs only encoding and transfer costs. Fitting lines to these runs makes it possible to separate the processing cost from encoding and transfer.

Figure 21 shows the VeloPix track forwarding algorithm's performance as time vs batch size with the data rates for runs spanning at least 20 batch sizes plotted over it. The runs split into two groups: those smaller than about 2.5 MB have data transfer rate of 0.23 GB/s, and those larger have the data transfer rate of 0.31 GB/s.

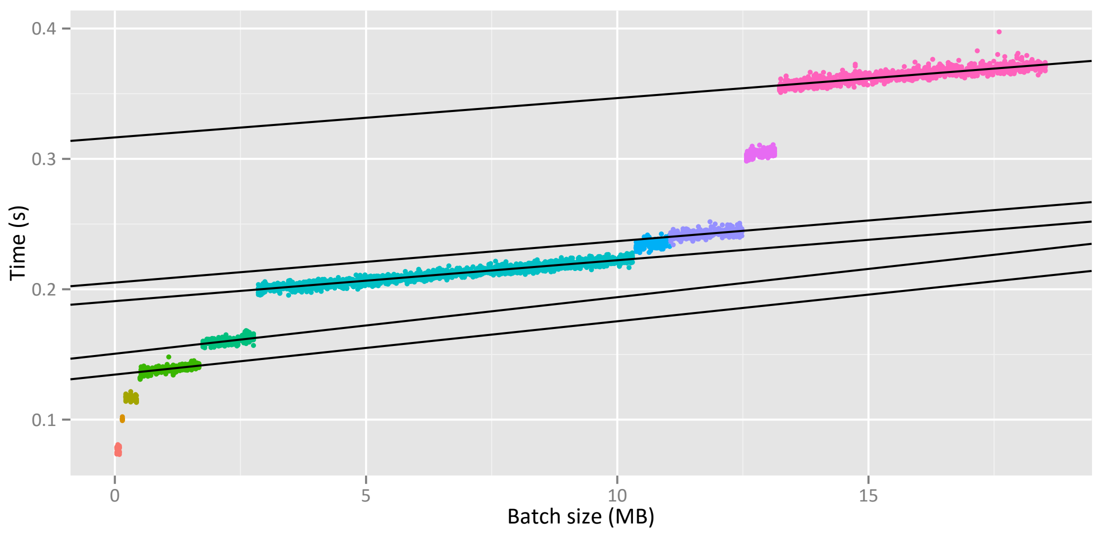

Figure 21 CPU-GPU transfer rates using the track forwarding algorithm with varying bach sizes.

Table 4 lists these numbers alongside Coprocessor Manager's latency and throughput from section 9.2. Events produced at LHCb are small compared to those of the other experiments — on the order of 100 KB, meaning that batches will be made up through many small transfers. We see that the added infrastructure's 50 µs latency is 3-4 orders of magnitude smaller than the algorithm's run time, making it negligible even when there are tens or hundreds of events in flight.

Coprocessor Manager latency	50 µs
Coprocessor Manager transfer	2.7-3.9 GB/s
CPU-GPU transfer	0.23-0.31 GB/s
GPU algorithm	76-316 ms

Table 4 Performance summary.

The cost of transferring data between processes is also an order of magnitude smaller than the cost of transferring it between the CPU and the GPU and is also small relative to the algorithm's run time. This means that the Coprocessor Manager can be used for GPU algorithms without incurring a significant performance penalty.

It should also be noted that VELO tracking is a relatively simple task, and data from other detectors, such as RICH, is likely to be even more expensive to reconstruct. It may prove necessary to gather data from multiple nodes, incurring higher transfer overhead. It may also be necessary to restrict batches to small numbers of events. In those cases, the GPU's efficiency can be raised by performing more computation on the data available. Performing full event reconstruction on the GPU instead of doing it separately for different detectors may prove to be more effective.

Finally, regarding LHCb's performance requirements, the 50 μs added by the Coprocessor manager are well within LHCb HLT's tolerance. The Coprocessor Manager's base transfer rate of 2.7-3.9 GB/s is well in excess of the 300-400 MB/s that a node can currently process. This means that inter-process communication is unlikely to be the bottleneck in the system. The CPU-GPU transfer is significantly slower. This cost could be masked by keeping the CPU busy while the transfer is taking place. Gaudi Hive could use this time to process other tasks.

10.2 Cost

A study by Daniel Cámpora and Cédric Potterat examines performance of an OpenCL port of the track forwarding algorithm on different platforms [81]. As described in section 4.4.1, OpenCL is an API for parallel computation, similar to CUDA. However, unlike CUDA, OpenCL is an open standard, so an OpenCL program can be executed on a large variety of hardware from different vendors.

The study ran a vectorized sequential version of the algorithm on an Intel Xeon E5-2630 v3 CPU using enough Gaudi instances to saturate all of the processor's 40 logical cores. It also ran a parallel OpenCL version on AMD HD 7970 and NVIDIA GTX 980 video cards, as well as an Intel Xeon Phi accelerator. Results of the study are summarized in Figure 22 as speedup of parallel versions compared to the sequential. Table 5 combines this data with the hardware prices current for November 2015 to calculate the throughput per dollar.

While this result presents a sample of only a single algorithm and a few different coprocessors, it suggests that GPUs could be sufficiently beneficial to the EFF upgrade to warrant further research.

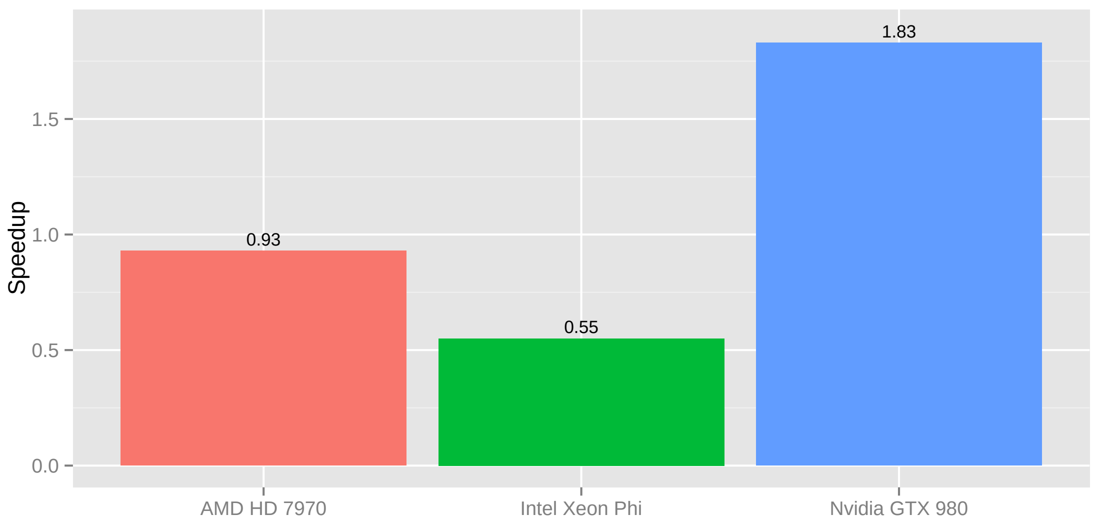

Figure 22 Cross-platform speed vs sequential; speed measured in events/s.

Processor	Relative throughput per US dollar
Intel Xeon CPU E5-2630 v3 @ 2.4 GHz	1.0 (= 1.00 · \$685.99/\$685.99)
Nvidia GTX 980	2.5 (= 1.83 · \$685.99/\$508.00)
AMD HD 7970	3.5 (= 0.93 · \$685.99/\$189.99)
Intel Xeon Phi	Preproduction

Table 5 Cost per throughput comparison.

Over the past years, GPU performance growth has greatly outpaced CPU performance growth for single-precision floating-point computation. More recently, GPUs have begun to pull ahead in double-precision computation, as well. Table 5 summarizes the performance figures as cited by the hardware manufacturers. While scientific computations tend to favour double precision, single-precision floating point is sufficient for some algorithms; the track-forwarding algorithm produces identical results with single and double precision. This means that the throughput/dollar metric may become even more favourable for GPUs in the coming years.

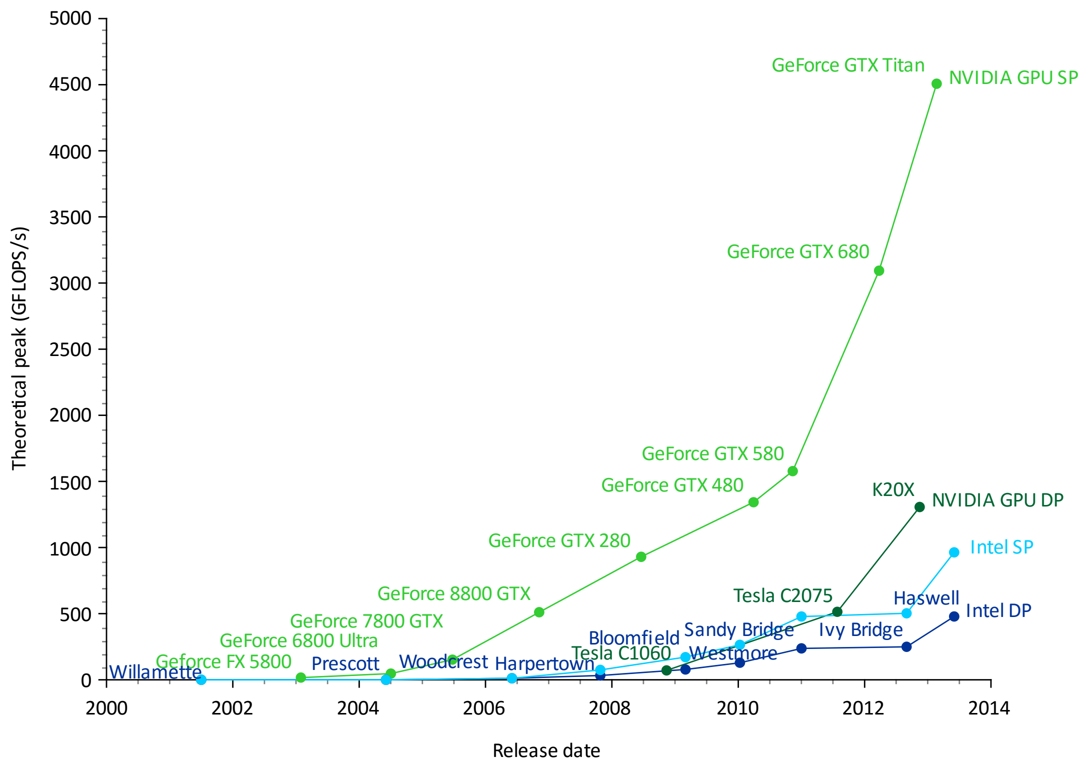

Figure 23 CPU vs GPU performance for single-precision (SP) and double-precision (DP) computation **[82]**.

11 Conclusion

The LHC is the most powerful particle collider ever built, with over four times the energy of the previous record holder – the Fermilab's Tevatron. LHC has already advanced our understanding of physics, but it does not stop there, as upgrades keep increasing its energy and luminosity. The experiments at the collider have to make sure that their detectors and data-gathering infrastructure make the best use of the collider upgrades.

The LHCb experiment is set to update its trigger infrastructure to process all the available data and to improve its flexibility by moving to a fully software trigger. There is a fixed budget for this upgrade, and the computer farm is limited in space, electric power, and cooling, which makes it essential to use the available resources efficiently. The upgraded computer farm should make better use of its hardware; at the same time, the possibility of using new kinds of coprocessors running new kinds of algorithms is being investigated.

The problem is that we cannot evaluate new kinds of coprocessors by simply adding them to the existing computer nodes. The existing infrastructure has to be extended to facilitate new algorithms that could make the best use of new hardware. This extension is necessary for both evaluating new code and for putting it into production.

We have developed the Coprocessor Manager as a way of integrating coprocessors and algorithms running on them with the existing infrastructure. The software allows algorithm development in the normal Gaudi environment and takes care of their hosting and scheduling. The Coprocessor Manager is capable of connecting to multiple concurrent event-processing threads and combining multiple calls to the same algorithm into batches. This can be essential to achieving high throughput with GPUs.

We have analysed the Coprocessor's performance and found that the added latency is three orders of magnitude below the run time for a well-optimized GPU implementation of a VELO tracking algorithm. The cost of transferring data to the Coprocessor Manager is also an order of magnitude smaller than the cost of moving data between the CPU and the GPU. We conclude that the overhead added by the Coprocessor Manager is small relative to the algorithm's performance.

In the end, there are two essential questions for considering coprocessor use in the coming LHCb Online upgrade: whether the new algorithms provide enough of a performance benefit per dollar to justify the added complexity, and whether we can write the required algorithms. So far, we see that GPUs can be as much as three times more efficient than typical Xeon CPUs. Since GPU performance growth has been outpacing CPU performance growth for years, the gap in throughput/dollar ratio may even widen in future.

The second question is equally interesting. In truth, it is not yet known which are the most important algorithms to rewrite for GPUs or even how many of them might be important to rewrite. There are few people available for this task, and those who are available are typically physicists rather than software developers. While Intel Xeon Phi and Nvidia CUDA make concerted efforts towards making coprocessor development as similar to normal Fortran or C++ development as possible, there is still a mental shift that is required for writing algorithms in a way that would perform well on massively parallel hardware. The developer has to have a deep understanding of the target hardware, of heterogeneous processing, and thread synchronization. A culture of writing parallel physics algorithms needs to develop.

It follows that we need to be developing new parallel algorithms and infrastructure not only to prepare for the coming upgrade, but also to learn, to find the hidden pitfalls, to learn the drawbacks and benefits of the technologies available to us.

Bibliography

[1] Intel Xeon Phi. [Online]. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

[2] (2015, November) The Green500 List - November 2015. [Online]. http://www.green500.org/lists/green201511

[3] Gianmaria Collazuol, Gianluca Lamanna, Felice Pantaleo, and Marco Sozzi, "Real-Time Use of GPUs in NA62 Experiment," in *Cellular Nanoscale Networks and Their Applications (CNNA)*, 2012.

[4] Sergey Gorbunov et al., "ALICE HLT high speed tracking on GPU," *Nuclear Science, IEEE Transactions on*, vol. 58, no. 4, pp. 1845--1851, 2012.

[5] Dmitry Emeliyanov and Jacob Howard, "GPU-based tracking for ATLAS Level 2 Trigger," ATLAS note 2012.

[6] Maik Dankel, Rene Boing, Jacob Howard, Sami Kama, and Matteo Bauce, "Use of hardware accelerators for ATLAS computing," ATL-COM-SOFT-2014-04, 2014.

[7] Sami Kama, "Triggering events with GPUs at ATLAS," CERN, Geneva, ATL-DAQ-PROC-2015-013, 2015.

[8] Alexey Badalov, Daniel Cámpora, Niko Neufeld, and Xavier Vilasís-Cardona, "LHCb GPU Acceleration Project," *Journal of Instrumentation*, vol. 11, no. 01, p. P01001, 2016.

[9] Alexey Badalov, Daniel Hugo Campora Perez, Alexander Zvyagin, Niko Neufeld, and Xavier Vilasis Cardona, "A GPU offloading mechanism for LHCb," in *Journal of Physics: Conference Series*, vol. 513, Amsterdam, 2014, p. 052004.

[10] N. Jarosik et al., "Seven-Year Wilkinson Microwave Anisotropy Probe (WMAP1) Observations: Sky Maps, Systematic Errors, and Basic Results," *ApJS*, p. 192, 2011.

[11] Lyndon Evans and Philip Bryant, "LHC Machine," *Journal of Instrumentation*, vol. 3, no. 08, 2008.

[12] J. J. Thomson, "XL. Cathode Rays," *Philosophical Magazine*, vol. 44, pp. 293-316, 1897.

[13] Ernest Rutherford, "The Scattering of α and β Particles by Matter and the Structure of the Atom," *Philosophical Magazine*, vol. 21, pp. 379-398, May

1911.

[14] Ernest Rutherford, "Collision of α Particles with Light Atoms IV. An Anomalous Effect in Nitrogen," *Philosophical Magazine*, vol. 37, p. 581, 1919.

[15] James Chadwick, "Possible Existence of a Neutron," *Nature*, vol. 129, no. 3252, p. 312, February 1932.

[16] Murray Gell-Mann, "A schematic model of baryons and mesons," *Physics Letters*, vol. 3, pp. 214-215, February 1964.

[17] Sheldon Lee Glashow, John Iliopoulos, and Luciano Maiani, "Weak Interactions with Lepton–Hadron Symmetry," *Physical Review*, vol. 2, no. 7, pp. 1285-1292, October 1970.

[18] Michael Riordan, *The Hunting of the Quark: A True Story of Modern Physics.*: Simon & Schuster, 1987, p. 210.

[19] Makoto Kobayashi and Toshihide Maskawa, "CP-violation in the renormalizable theory of weak interaction," *Progress of Theoretical Physics*, vol. 49, no. 2, pp. 652-657, 1973.

[20] S. W. Herb et al., "Observation of a Dimuon Resonance at 9.5 GeV in 400-GeV Proton-Nucleus Collisions," *Physical Review Letters*, vol. 39, no. 5, pp. 252-255, August 1977.

[21] CDF Collaboration, "Observation of top quark production in p p collisions with the collider detector at fermilab," *Physical Review Letters*, vol. 74, pp. 2626-2631, 1995.

[22] DØ Collaboration, "Search for High Mass Top Quark Production in pp^{-} Collisions at s = 1.8 TeV," *Physical Review Letters*, vol. 74, pp. 2422-2426, 1995.

[23] LHCb Collaboration, *LHCb: Technical Proposal*. Geneva: CERN, 1998.

[24] LHCb Collaboration, "The LHCb Detector at the LHC," *Journal of Instrumentation*, vol. 3, 2008.

[25] LHCb Collaboration, "LHCb magnet: Technical Design Report," CERN, Geneva, 2000.

[26] LHCb Collaboration, "LHCb VELO (VErtex LOcator): Technical Design Report," CERN, Geneva, 2001.

[27] LHCb Collaboration, "Letter of intent for the LHCb upgrade," CERN, Geneva, Technical Design Report 2011.

[28] LHCb Collaboration, "LHCb Reoptimized Detector Design and Performance,"

CERN, Geneva, Technical Design Report CERN-LHCC-2003-030 ; LHCb-TDR-9, 2003.

[29] LHCb Collaboration, "LHCb calorimeters : Technical Design Report," CERN, Geneva, Technical Design Report CERN-LHCC-2000-0036-[sic!] ; CERN-LHCC-2000-036 ; LHCb-TDR-2, 2000.

[30] LHCb Collaboration, "LHCb muon system: Technical Design Report," CERN, Geneva, 2001.

[31] LHCb Collaboration, "LHCb VELO Upgrade Technical Design Report," CERN, Geneva, Technical Design Report CERN-LHCC-2013-021. LHCB-TDR-013, 2013.

[32] LHCb Collaboration, "LHCb online system Technical Design Report: Data acquisition and experiment control.," *Work*, vol. 501, 2001.

[33] (20155, September) LHCb improves trigger in Run 2. [Online]. http://cerncourier.com/cws/article/cern/62495

[34] (2015, October) The LHCb Event Filter Farm (EFF). [Online]. https://lbonupgrade.cern.ch/doku.php?id=outreach:the_lhcb_eventfilterfarm

[35] LHCb Collaboration, *LHCb computing: Technical Design Report*. Geneva: CERN, 2005.

[36] Mark Cattaneo, " GAUDI - The Software Architecture and Framework for building LHCb data processing applications ," in *International Conference on Computing in High Energy and Nuclear Physics*, February 2000.

[37] P. Mato, "GAUDI — Architecture design document," CERN, Geneva, LHCb-98-064, 1998.

[38] R., Antunes, A. Nobrega et al., "LHCb computing technical design report," 2005.

[39] Presentations about LHCb reconstruction. [Online]. http://lhcb-comp.web.cern.ch/lhcb-comp/Reconstruction/Talks/Talks.htm

[40] Marco Cattaneo, "Event Reconstruction for LHCb," in *2nd LHCb software week*, Geneva, 1999.

[41] (2015) The LHCb Event Model. [Online]. https://twiki.cern.ch/twiki/bin/view/LHCb/LHCbEventModel

[42] Sébastien Ponce, Ivan Belyaev, Pere Mato Vila, and Andrea Valassi, "Detector description framework in LHCb," *arXiv preprint physics*, 2003.

[43] Python. [Online]. https://www.python.org/

[44] Marco Clemencic and P. Mato, "A CMake-based build and configuration framework," *J. Phys.: Conf. Ser.*, vol. 396, p. 11, 2012.

[45] Christian Arnault, "CMT: A software confriguraiton management tool," *Proceeding of CHEP2000*, 2000.

[46] CMake - Cross Platform Make. [Online]. http://www.cmake.org

[47] Marco Cattaneo and LHCb event model working group, "The new LHCb Event Data Model," CERN, Geneva, LHCb Technical Note LHCb 2001-142, 2016.

[48] I. Belyaev et al., "Simulation application for the LHCb experiment," in *Proceedings of CHEP03*, San Diego, 2003.

[49] S. Agostinelli et al., "GEANT4—a simulation toolkit," *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, pp. 250-303, 2003.

[50] Geant4 homepage. [Online]. https://geant4.web.cern.ch/geant4/

[51] Marco Clemencic, Benedikt Hegner, Pere Mato, and Danilo Piparo, "Preparing HEP software for concurrency," in *Journal of Physics: Conference Series*, vol. 513, Amsterdam, Netherlands, 2013.

[52] B. Hegner, "Running Concurrent Gaudi in Real Life: Status Update on MiniBrunel," CERN, ATLAS S&C Workshop Presentation 2013.

[53] Illya Shapoval, "Adaptive Scheduling Applied to Non-Deterministic Networks of Heterogeneous Tasks for Peak Throughput in Concurrent Gaudi," INFN, Ferrara, PhD Thesis 2016.

[54] Loïc Brarda et al., "A new data-centre for the LHCb experiment," *Journal of Physics: Conference series*, vol. 396, no. 1, 2012.

[55] Herb Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, vol. 30, no. 3, pp. 202-210, 2005.

[56] G. M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities.," in *AFIPS Conference Proceedings*, vol. 30, Atlantic City, N.J., 1967, pp. 483-485.

[57] OpenCL. [Online]. https://www.khronos.org/opencl/

[58] DirectX 12. [Online]. https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121%28v=vs.85%29.aspx

[59] Metal. [Online]. https://developer.apple.com/metal/

[60] Tomasz Bawej et al., "Boosting Event Building Performance using Infiniband

FDR," in *PoS TIPP2014*, vol. 2014, 2014, p. 190.

[61] Guoming Liu and Niko Neufeld, "DAQ architecture for the LHCb upgrade," *Journal of Physics: Conference Series*, vol. 513, pp. 12-27, 2014.

[62] Roberto Ammendola et al., "NaNet: a low-latency NIC enabling GPU-based real-time low level trigger systems.," *J. Phys.: Conf. Ser.*, vol. 513, p. 012018. 7 p, 2013.

[63] R. Ammendola et al., "GPU-based Real-time Triggering in the NA62 Experiment," CERN, Preprint arXiv:1606.04099, 2016.

[64] R. Ammendola et al., "APEnet+: a 3D Torus network optimized for GPU-based HPC Systems," *Journal of Physics: Conference Series*, vol. 396, p. 042059, 2012.

[65] The ALICE Collaboration, "The ALICE experiment at the CERN LHC," *JINST*, vol. 3, 2008.

[66] Rene Böing, Sebastian Fleischmann, Maik Dankel, and Peter Mättig, "Application of GPUs in ATLAS offline tracking," ATLAS, Computing workshop presentation 2014.

[67] Roberto Agostino Vitillo. (2016, January) YAMPL. [Online]. https://github.com/vitillo/yampl

[68] ØMQ. [Online]. http://zeromq.org/

[69] (2016) Apache Thrift. [Online]. https://thrift.apache.org/

[70] Protocol Buffers. [Online]. https://developers.google.com/protocol-buffers/

[71] Cap'n Proto. [Online]. https://capnproto.org/

[72] Illya Shapoval, "Predictive scheduling for low throughput in Gaudi Hive," in *6th LHCb Computer Workshop*, Paris, 2015.

[73] Ilya Shapoval, "Gaudi Hive on the Landscape of Heterogeneous Computing," in *FCPMF*, 2015.

[74] P. Gandini, C. Hadjivasiliou, and J. Wang, "Upgrade tracking with the UT Hits," CERN, Geneva, Technical report LHCb-PUB-2014-004, 2014.

[75] Rudolph Emil Kalman, "A new approach to linear filtering and prediction problems.," *Journal of Fluids Engineering*, vol. 82, pp. 35-45, 1960.

[76] Jeroen Ashwin Niels van Tilburg, "Track simulation and reconstruction in LHCb," University of Amsterdam, Amsterdam, PhD Thesis 2005.

[77] LHCb Collaboration, "Measurement of the track reconstruction efficiency at

LHCb," *Journal of Instrumentation*, vol. 10, no. 02, p. P02007, 2015.

[78] Daniel Hugo Cámpora Pérez, "A Study of a Parallel Implementation for the Pixel VELO subdetector," Universidad de Sevilla, Sevilla, MSc Thesis 2013.

[79] Daniel Hugo Cámpora Pérez, "A Study of a Parallel Implementation for the Pixel VELO Subdetector," Universidad de Sevilla, Sevilla, MSc thesis 2013.

[80] Olivier Callot, "FastVelo, a fast and efficient pattern recognition package for the Velo," 2011.

[81] Daniel Hugo Cámpora Pérez and Cédric Potterat, "Once and for all: OpenCL velopix cross-platform studies," in *6th LHCb Computing Workshop*, Paris, 2015.

[82] Michael Galloy. (2013, June) CPU vs GPU performance. [Online]. http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html

[83] Michel De Cian, "Track Reconstruction Efficiency and Analysis of $B^0 \to K^{*0} \mu^+ \mu^-$," Universtität Zürich, Zurich, PhD Thesis 2013.

[84] P. R Barbosa-Marinho, "LHCb online system Technical Design Report: Data acquisition and experiment control.," *Work*, vol. 501, 2001.