# LHCb GPU Acceleration Project

## Public Note


Issue: 1
Revision: 1

Reference: LHCb-PUB-2015-021
Created: September 4, 2015
Last modified: October 12, 2015



**Prepared By:** Alexey Badalov[a], Daniel Cámpora[b], Niko Neufeld[b], Xavier Vilasís-Cardona[a]

[a] La Salle – Ramon Llull University, Spain
[b] CERN, Switzerland



LHCb-PUB-2015-021
04/09/2015


# Abstract

The LHCb detector is due to be upgraded for processing high-luminosity collisions, which will increase the load on its computation infrastructure from 100 GB/s to 4 TB/s, encouraging us to look for new ways of accelerating the Online reconstruction. The Coprocessor Manager is our new framework for integrating LHCb's existing computation pipelines with massively parallel algorithms running on GPUs and other accelerators. This paper describes the system and analyzes its performance.

# Document Status Sheet

| 1. Document Title: LHCb GPU Acceleration Project | | | |
|---|---|---|---|
| 2. Document Reference Number: LHCb-PUB-2015-021 | | | |
| **3. Issue** | **4. Revision** | **5. Date** | **6. Reason for change** |
| Final | 1 | October 5, 2015 | First version |
| Final | 2 | October 12, 2015 | Fixed a typo in batching performance numbers.<br>Expanded acknowledgements. |

# Table of Contents

**1 Introduction** ..... **1**

**2 Architecture design** ..... **1**

**3 System overview** ..... **2**

1.1 Introduction ..... 2
1.2 Gaudi framework integration ..... 3
3.1.1. CpService ..... 3
3.1.2. CpAlgorithm ..... 4
1.3 cpserver and cpdriver ..... 5
1.4 Client-server architecture ..... 6
1.5 Concurrent Gaudi considerations ..... 7

**4 VELO tracking on the GPU** ..... **8**

1.6 VELO Pixel ..... 8
1.7 Massively parallel VELO Pixel tracking ..... 9

**5 Performance evaluation** ..... **9**

1.8 Hardware setup ..... 9
1.9 Overhead analysis ..... 9
1.10 Batching performance ..... 10

**6 Conclusions** ..... **12**

**7 Acknowledgements** ..... **12**

**8 References** ..... **13**

# List of Figures

Figure 1. Coprocessor Manager architecture. ..... 7

Figure 2 Schematic layout of the upgraded VELO. ..... 8

Figure 3. *cpserver* data transfer overhead in grey (above) and baseline network transfer rate in red (below) along with the trend lines. ..... 10

Figure 4. Times over 10 runs for VELO Pixel tracking running on *cpserver* with varying batch sizes: full dataset above and a zoomed in view below. Run data points are gray, median values black. ..... 11

# List of Tables

Table 1. Comparison of a CPU and a GPU VELO Pixel tracking algorithm. ..... 12

# 1 Introduction

The LHCb experiment [1] [2], located at Point 8 of the LHC, is a single-arm spectrometer with a forward geometry. The detector's hardware is being upgraded with the intent of sustaining luminosities of up to $2 \times 10^{33}\text{cm}^{-2}\text{s}^{-1}$ [3] from current $4 \times 10^{32}\text{cm}^{-2}\text{s}^{-1}$ [4]. These higher luminosities will increase the number of particle collisions at the detector and require dramatic increases in the data processing capacity. LHCb's on-site data-centre is comprised of 27,040 physical CPU cores; increasing this number, as well as each core's individual performance, is necessary, but more cost-effective approaches are highly desirable. This motivates the search for innovative ways of dealing with increased data loads [5].

One alternative to the straightforward expansion of the current infrastructure is to shift a part of the load to GPU processors. GPUs often achieve higher data throughput than conventional processors while drawing less power — as evidenced by their use in nine out of the ten top supercomputers in the November 2014 Green500 list [6]. However, the algorithms currently in use are optimized for serial execution; the new massively parallel accelerators require these algorithms to be redesigned or replaced with entirely new approaches. Massively parallel processing also calls for changes in the way data travels through the computation infrastructure.

We describe the extension of the existing system that we have created, which allows using new massively parallel algorithms in combination with existing proven software, called a Coprocessor Manager. It creates an environment for adding new algorithms to existing computation pipelines and manages the flow of data in a way that lets the algorithms use the hardware most effectively. The extension's design is informed by the lessons learned from the other experiments' efforts [7] [8] [9] and makes use of a distinct architecture.

The rest of the paper is organized as follows: Chapter 2 compares Coprocessor Manager's architecture to other similar projects; Chapter 3 provides a detailed description of the Coprocessor Manager; Chapter 4 describes a tracking algorithm we used to test it; Chapter 5 evaluates the system's performance; and Chapter 6 summarizes the results.

# 2 Architecture design

Other collider experiments have investigated the idea of moving some of their computation to GPUs. We document how these efforts, ranging from implementation of specialized algorithms to creation of new software infrastructure, informed Coprocessor Manager's design.

Researchers at the NA62 experiment have investigated the problem of adapting specific algorithms to massively parallel architectures [7]. Using Nvidia's CUDA [10] GPU programming platform, they make the important observation that the overhead cost of setting up the computation is high compared to the cost of processing a single particle collision event — this makes it very desirable to process events in batches, amortizing the setup cost.

Researchers at ALICE (A Large Ion Collider Experiment) at LHC took the next step and investigated the issues that arise from GPU algorithm integration with existing software infrastructure [8]. They have successfully added GPU track reconstruction to ALICE's High-Level Trigger (HLT) and the offline reconstruction framework. This turned out to require adjustments in the HLT framework, as well as AliRoot — ALICE's offline software framework for data analysis, event reconstruction, and simulation. 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. The authors also encountered a mismatch in the threading models used by the HLT framework and CUDA. The mismatch was that HLT processes events over multiple asynchronous concurrent threads, while CUDA is intended for single-threaded use.

ATLAS (A Toroidal LHC Apparatus) has mounted a serious effort to integrate GPU computation with its software infrastructure [9]. ATLAS has a special relationship with LHCb in that its computation framework

— Athena — is an adaptation of LHCb's. Like Gaudi, Athena spawns multiple concurrent processes for multiple events. ATLAS's initial design was to make remote procedure calls to the GPU. In this design, a separate *GPU Server* process hosts a number of GPU kernels. Athena processes connect to this process, choose a kernel, send parameters to the kernel, and wait to receive the results. The server accepts requests and calls whichever kernel is requested in the order it receives the requests.

This approach is in practice very similar to 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 cleaner separation between regular Athena pipeline code and CUDA-dependent algorithms; it also has the added benefit of allowing remote calls to CUDA kernels to be made across a network. Finally, hosting all the kernels in a separate long-lived process could potentially save on the costs 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.

ALICE's and NA62's experience with GPUs shows the difficulty in maintaining high utilization of the processing units. Detectors like LHCb, where individual events require relatively little computation, need massively parallel event processing systems to combine and simultaneously process multiple events. This difficulty will increase as long as the growth of GPU performance outpaces growth in information volume produced by particle detectors.

Another issue that has to be addressed is the system's ease of use by the algorithm developer. We tap into an existing body of research on GPU algorithms for high-energy physics. If it happens to be difficult to transfer an algorithm's implementation from a standalone version to the detector's production system, our choice of massively parallel algorithms may be restricted.

The Coprocessor Manager addresses the shortcomings of its predecessors. It is expressly designed to combine data from multiple sources — a *coprocessor* algorithm is an object that takes a batch of objects of a given type and creates an output for each of these objects. Special provisions are made for error handling. The way an algorithm performs its computations, what libraries and devices it uses, is left up to its author. This lack of restrictions serves to facilitate porting of existing tested code to the new infrastructure.

# 3 System overview

## 1.1 Introduction

Any system for running algorithms within LHCb's software environment has to adapt to Gaudi. Gaudi [11] [12] is LHCb's software framework hosting a wide range of physics data processing applications, from simulation to reconstruction and analysis; all of these application are built as Gaudi components. Experiment-specific software, such as the Event Model [13], which describes the event data classes and their relationships, and the Detector Description [14], 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.

The Gaudi framework was originally designed and implemented in the 1990s, but consumer multicore CPUs did not become widespread until 2006 with the release of Intel Core 2 Duo, and general-purpose GPU computation later yet. It builds on the idea of having each node process events one by one; several Gaudi instances can run concurrently on a single multicore machine as multiple processes, but each instance has its own copies of the transient stores and they do not communicate with each other. This approach to

processing events was prudent when the framework was designed, but in the present it stands in the way of making effective use of powerful massively parallel hardware, which is typically heavily optimized for single-instruction multiple-data (SIMD) execution. SIMD execution favours applying the same algorithm to multiple events at the same time, processing them in batches.

The Coprocessor Manager is the external process and accompanying infrastructure that enables Gaudi algorithms to leverage the power of massively parallel algorithm accelerators, such as GPUs without changing the fundamental nature of the framework. It uses a client/server architecture, where a process called *cpserver* runs on each coprocessor-equipped machine, while multiple Gaudi instances on the same machine or on the same network connect to it as clients. The *cpserver* process hosts all of the GPU algorithms. When a client sends data for processing, it specifies which algorithm is to be used for the task. As multiple clients send data to *cpserver* concurrently, it schedules them in a way that maintains high throughput, as explained in more detail in section 1.4.

The Coprocessor Manager additionally offers facilities for development and fine-tuning of massively parallel algorithms. The *cpserver* process logs algorithm execution times and data batch sizes. It is also capable of recording input and output for each client to disk for later playback and verification using the *cpdriver tool*. This way, the same data can be played back repeatedly and without waiting for it being generated by Gaudi. We use this for analyzing algorithm performance and maintaining correctness.

## 1.2 Gaudi framework integration

The Coprocessor Manager extends the Gaudi framework and connects to it at two points. First, it offers a Gaudi service that allows existing algorithms to exchange data with GPU algorithms, although it should be noted that the Coprocessor Manager can also exchange data with sources other than Gaudi. Second, it exposes an extension point that allows GPU algorithms to be developed within Gaudi alongside the existing software and be picked up by *cpserver* automatically. We now take a more detailed look.

### 3.1.1. CpService

Algorithms running on Gaudi have access to various services. The Coprocessor Manager exposes a service of its own via the *ICpService* interface, which can be instantiated in the same way as any other Gaudi service:

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

*ICpService* exposes only a single function: *submitData*. This function takes a data array and the name of the GPU algorithm to be used to process it, 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);
};
```

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);
```

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();
}
```

### 3.1.2. CpAlgorithm

The other point of integration into the Gaudi framework consists of allowing GPU algorithms to be developed alongside the rest of Gaudi components and added without the need to modify *cpserver*.

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);
};
```

Coprocessor Manager's key feature 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*. It could process these arrays one by one or, to make optimal use of the GPU, it could process them all together.

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 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 algorithm needs to call the *DECLARE_COMPONENT* macro with the name of the implementation class. Here is 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);

  // allocate memory for the output and copy (this could be optimized)
  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);
  }
}
```

## 1.3 cpserver and cpdriver

Coprocessor Manager includes two important executables: *cpserver* and *cpdriver*. The former executable implements the service that hosts algorithms and communicates with Gaudi or other data sources; the latter is a utility that can replay pre-recorded data to *cpserver* and verify the results. *cpdriver* is an example of sending data to *cpserver* from a source other than Gaudi.

*cpserver* can be used for launching a new instance of the server process, as well as controlling one that is already running. It takes the following arguments:

**daemonize** – runs *cpserver* as a daemon. Otherwise, it is run as a regular console process.

**exit** – terminates a running instance. The instance is identified by the *service* argument.

**load** – loads an algorithm into a running instance. For the algorithm used in a previous example, the algorithm name would be *PrPixelCudaAlgorithm*.

**connection** – *local* or *tcp*; toggles the means of communication with *cpserver*.

**localPath** – a local Unix socket path. This is a high-performance means of communication within a single machine.

**tcpHost** – a network socket address. Used for communication across a network.

**tcpPort** – a network socket port.

**datadir** – enables recording of input and output data to disk. Specifies the directory, in which to store the files. The directory should exist.

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 call to *submitData* generates a file containing the data received and the data produced by the algorithm used to process it. These files can then be played back using *cpdriver*. It 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 contamination by processes producing the data. It is also useful for maintaining algorithm correctness: running a previously recorded data set with the *verify* flag after modifying an algorithm to make sure that it produces the same output.

The LHCb code is organized into several large collections, each containing the software specialized for some task. The Coprocessor Manager can be used with any of these collections. Here is sample usage with event reconstruction in Brunel (B Reconstruction, UNderstanding Events in Lhcb) [15] [16]:

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

# 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
```

## 1.4 Client-server architecture

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. A custom library called *CpIpc* handles the communications.

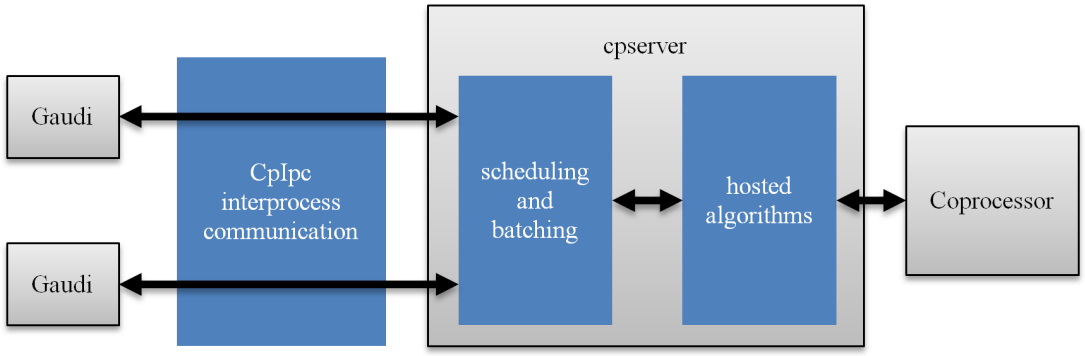


Figure 1. Coprocessor Manager architecture.

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 goes 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 are done on a dedicated thread. Algorithms are executed one after another whenever data is available. The scheduling algorithm is a variant of the first-come-first-serve (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.

The communication protocol and network machinery are encapsulated in the *CpIpc* library. The server uses this library to communicate with its clients and with other instances of itself, when the *cpserver* executable is used to terminate a running 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.

## 1.5 Concurrent Gaudi considerations

There is an ongoing effort to add support for parallelism inside the Gaudi framework [17]. The prototype implementation of Gaudi with added task parallelism is called GaudiHive. It uses the Intel Thread Building Blocks to divide work into tasks and its own scheduler to run them.

GaudiHive could become a more efficient way of sending concurrent requests to the Coprocessor Manager; the more concurrent requests the Coprocessor Manager receives, the more efficiently it can schedule their execution. However, having multiple schedulers, one in *cpserver* and one in GaudiHive, creates an

additional difficulty: a task waiting for *ICpService::submitData* to return would stall GaudiHive's execution queue due to GaudiHive having no concept of idle waiting. In order to cooperate with the Coprocessor Manager, GaudiHive needs a way for tasks to suspend themselves. We currently do not know whether it will gain this capability, but this is an important consideration when adding parallelism to computation frameworks.

# 4 VELO tracking on the GPU

## 1.6 VELO Pixel

We evaluate the Coprocessor Manager's performance using a GPU-based VELO Pixel tracking algorithm. In this chapter we describe the algorithm and the system it measures.

The VErtex LOcator (VELO) is a silicon strip detector that is the part of the LHCb detector closest to the interaction region [18]. Its job is to measure the 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.

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 [19]. VELO Pixel will consist of 52 modules placed along the z-axis, each featuring 12 radiation-hard VeloPix ASIC chips with 256x256-pixel matrices. 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.

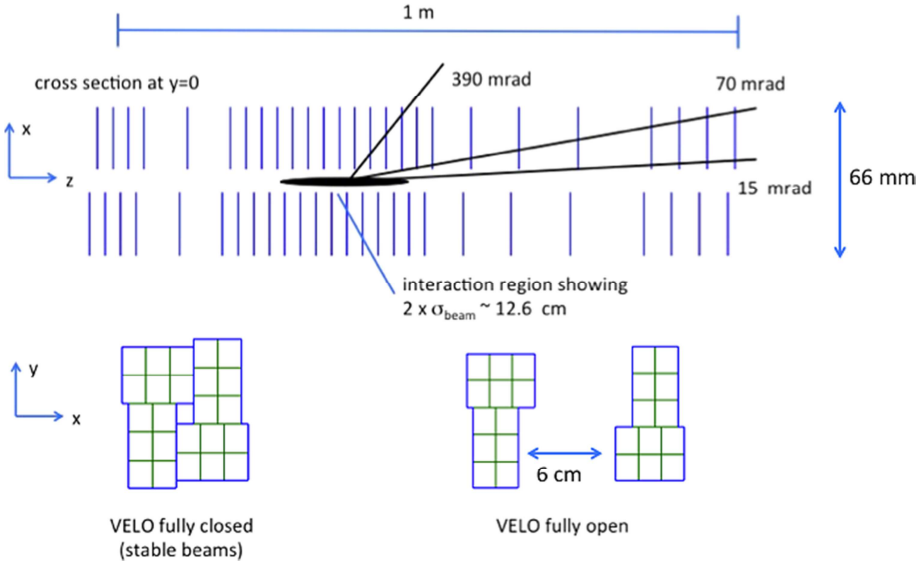


Figure 2 Schematic layout of the upgraded VELO.

Track reconstruction takes hit $(x, y, z)$ coordinates produced by the detector and computes track state vectors of the form:

$$TrackState(z) = (x \quad y \quad t_x \quad t_y \quad q/p)$$

Variables $t_x = \partial x/\partial z$ and $t_y = \partial y/\partial z$ are the track slopes in the xz and yz plane. The last parameter $q/p$ is an estimate charge divided by momentum for the reconstructed particle, used to reconstruct trajectories in a magnetic field.

## 1.7 Massively parallel VELO Pixel tracking

We ran our tests on a reconstruction algorithm 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 is given hit coordinates in batches from multiple events and for each event compiles lists of hits likely to form tracks. The algorithm 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. The preparation step takes $O(n^2)$ time, where $n$ is the average number of hits per module. Because some of the steps are interlaced and because we cannot accurately time partial CUDA kernel execution, we do not measure the exact contribution of each step.

The track-building step is based on the idea of track forwarding. A track is forwarded for further processing 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. This takes $O(n^2)$ time.

For every sensor, the algorithm also examines every hit registered 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 compatible is added to the tracks being forwarded. This operation takes $O(n^3)$ time.

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.

# 5 Performance evaluation

## 1.8 Hardware setup

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

## 1.9 Overhead analysis

Compared to processing data directly in Gaudi, the Coprocessor Manager introduces some additional overhead. Data sent to it needs to be serialized and responses from it deserialized. Serialization is consolidation of data structures into continuous blocks with no references to external data, while deserialization is the reverse process. Additionally, there is some overhead for data transmission, batching, and algorithm execution scheduling.

These additional tasks are performed on the CPU. Ideally, the CPU would be serving new data to the Coprocessor Manager while old data is being processed. In this case, there would be no penalty to the throughput. However, it could also happen that the CPU would stall while waiting for the coprocessor, and in this case we should hope that the combined overhead and the hosted algorithm's execution time are still smaller than the CPU algorithm execution time.

The overhead induced by the Coprocessor Manager also introduces additional latency. To measure this we use a simple data algorithm that does not perform any computation on the data it receives and send it varying amounts of random bytes via *cpdriver*. Communication via local Unix sockets provides fast inter-process communication similar in performance to shared memory. We conducted an experiment by sending 1000 data packets ranging from 0 to 500,000 bytes using the Coprocessor Manager and a minimal application using the same network interface as a baseline. Figure 3 shows the results.

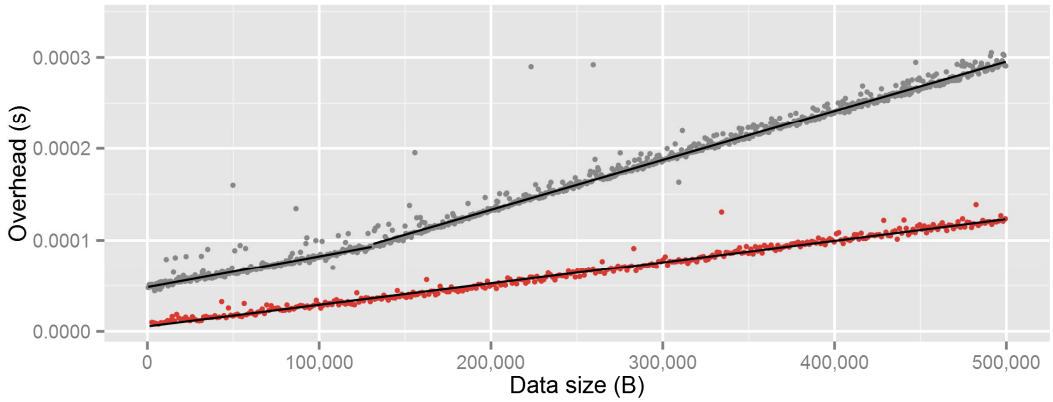


Figure 3. *cpserver* data transfer overhead in grey (above) and baseline network transfer rate in red (below) along with the trend lines.

There is a simple linear relationship between data size and time. 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 coded into the Coprocessor Manager and may be hardware-dependent.

Events produced at LHCb are small compared to those of the other experiments — on the order of 100 KB, and so benefit from the faster transfer rate. The algorithms that run on a GPU can take on the order of milliseconds to hundreds of milliseconds to execute, so the communication overhead is negligible. The added 50 µs latency is also well within the LHCb High Level Trigger's tolerance.

## 1.10 Batching performance

We ran the tracking algorithm on Monte-Carlo simulated hit data, controlling the size of batches, in which the CUDA *PrPixel* coprocessor algorithm was given event data. Performance data from 10 runs is summarized in Figure 4.

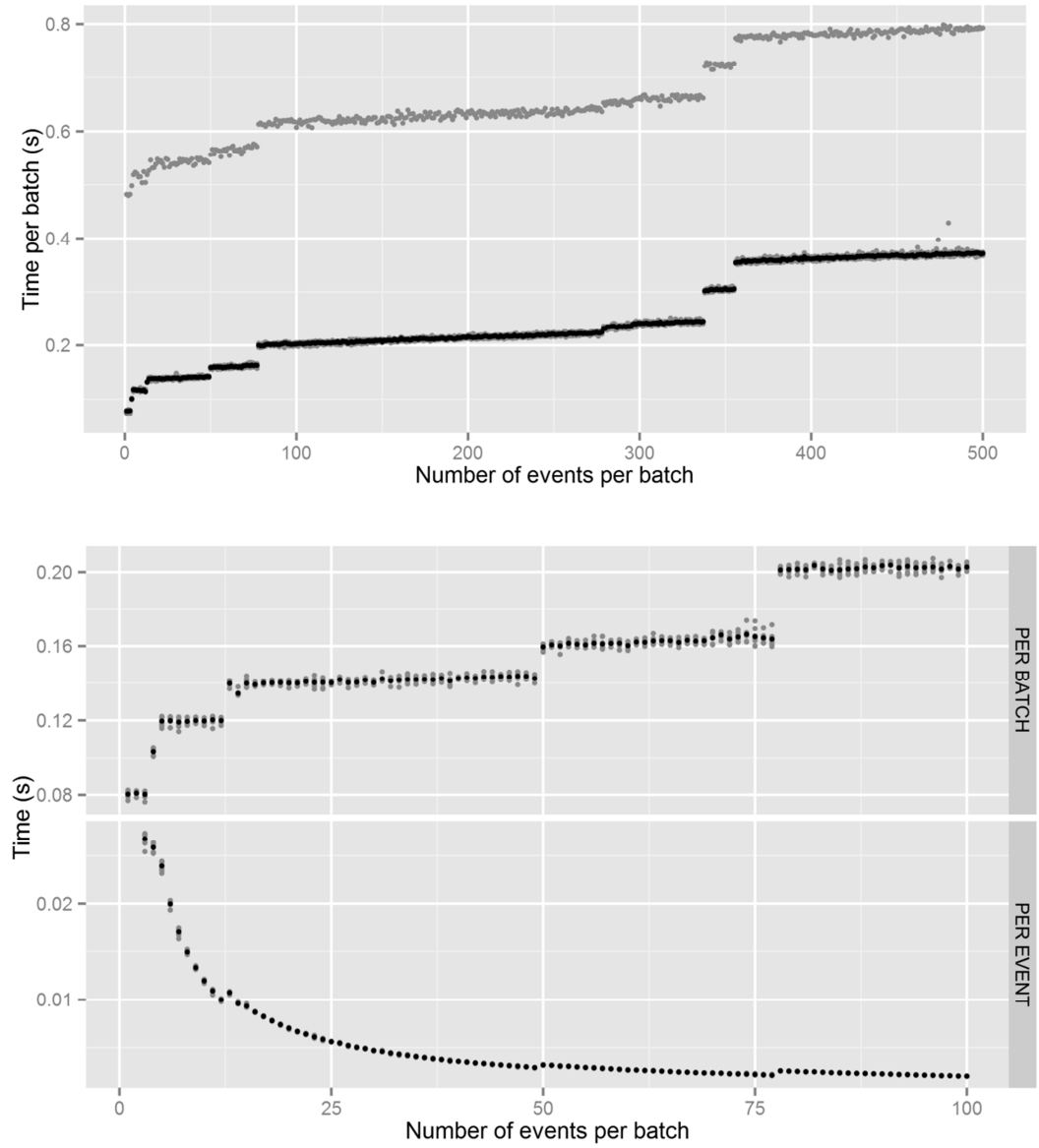


Figure 4. Times over 10 runs for VELO Pixel tracking running on *cpserver* with varying batch sizes: full dataset above and a zoomed in view below. Run data points are gray, median values black.

We ran tests for batches of up to 500 events. *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 reach the current CUDA VELO Pixel tracking algorithm's implementation's batch size limit.

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 want to process data in the largest batches that can be processed within exceeding latency tolerances.

We compare our algorithm with an up-to-date version of PrPixel — the CPU-based VeloPix tracking algorithm [19]. Table 1 makes the comparison in terms of time, ghost rate, and efficiency; a ghost track is one that cannot be associated to any simulated particle; 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 | Track forwarding | |
|---|---|---|---|
| Time per event (ms) | 3.6 | 26.2 | batch of 1 |
| | | 3.5 | batch of 40 |
| Ghost rate | 1.7% | 0.8% | |
| Efficiency for long tracks | 98.3% | 98.0% | |
| Efficiency for long tracks over 5 GeV | 98.8% | 98.4% | |

Table 1. Comparison of a CPU and a GPU VELO Pixel tracking algorithm.

# 6 Conclusions

The Gaudi computation framework is designed for running multiple serial computations on fast multi-core CPUs. However, modern massively parallel coprocessors, such as GPUs, leverage large-scale vectorization to achieve their high throughput and energy efficiency. We have developed the Coprocessor Manager to add support for such processors to the current infrastructure and enable computation pipelines to mix existing algorithms with new optimized for massively parallel hardware.

We designed the Coprocessor Manager based on the results of other teams' investigations into bringing massively parallel computation to high-energy physics, mindful of our obligation to provide tools for algorithm development and support, as well as the special requirements of the new algorithms.

Coprocessor Manager's main distinguishing features is its ability to bring pieces of data from multiple sources together and process them in batches using specially designed algorithms running on massively parallel processors. In our tests we study one such algorithm and find that its performance is critically reliant on having large batches of hundreds of events to process.

We also studied the overhead incurred by having a separate process perform part of the computation. Our tests show that when Gaudi clients run on the same machine as the *cpserver* process, the overhead is small compared to the total processing time. There is a one-time cost of 0.4 s per *cpserver* instance for setting up the GPU and a constant 50 μs latency overhead due to the inter-process communication. Without algorithm-specific overhead, *cpserver* has a throughput of 1.7-2.7 GB/s. This overhead is within tolerance for LHCb's HLT trigger.

For the next step, we intend to test the Coprocessor Manager in a production environment.

# 7 Acknowledgements

Thanks to Conor Fitzpatrick, Manuel Tobias Schiller, and Vladimir Gligorov for reviewing the manuscript and offering their comments.

Thanks to Marco Corvo for implementing a TCP/IP communications layer, Mikhail Belous for feedback and testing, and Stefano Gallorini for feedback and discussions.

This work is supported by the Spanish MINECO (FPA2011- 30163 and CPAN CSD2007-00042 of the Programme Consolider-Ingenio 2010).

# 8 References

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

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

[3] The LHCb Collaboration, "LHCb Trigger and Online Upgrade Technical Design Report," CERN, CERN-LHCC-2014-016, 2014.

[4] The LHCb Collaboration, "LHCb Detector Performance," *International Journal of Modern Physics*, vol. A30, no. 07, 2015.

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

[6] (2014, November) The Green500 List - November 2014. [Online]. http://www.green500.org/lists/green201411

[7] 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.

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

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

[10] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40-53, 2008.

[11] 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.

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

[13] (2015) The LHCb Event Model. [Online]. https://twiki.cern.ch/twiki/bin/view/LHCb/LHCbEventModel

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

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

[16] Marco Cattaneo, "Event Reconstruction for LHCb," Presentation 1999.

[17] 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.

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

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