



university of
 groningen

faculty of science
and engineering

van swinderen institute for
particle physics and gravity

MASTER'S THESIS

EXPLORING GPU-COMPUTING TO ACCELERATE VERTEX FINDING IN PARTICLE PHYSICS

Author:

J. HEIT

Supervisors:

Dr. Ir. C.J.G. ONDERWATER

Dr. J.G. MESSCHENDORP

MSC. APPLIED PHYSICS
FACULTY OF MATHEMATICS & NATURAL SCIENCES
UNIVERSITY OF GRONINGEN

JULY 13, 2017

Contents

| | | |
|----------|---------------------------------------------|-----------|
| 1 | Introduction | 4 |
| 2 | LHCb Detector | 6 |
| 2.1 | Introduction | 6 |
| 2.2 | Vertex Locator (VELO) | 6 |
| 2.3 | RICH 1 and 2 | 8 |
| 2.4 | Magnet | 9 |
| 2.5 | Calorimeter | 9 |
| 2.6 | Muon Detector | 10 |
| 3 | Vertex Location | 11 |
| 3.1 | Technology | 11 |
| 3.2 | Tracking | 13 |
| 3.3 | Primary Vertices | 13 |
| 3.3.1 | Seeding | 14 |
| 3.3.2 | Fitting | 15 |
| 3.4 | Secondary Vertices | 15 |
| 4 | Motivation | 16 |
| 4.1 | Current Affairs | 16 |
| 4.2 | Upgrade | 16 |
| 4.2.1 | Future Performance | 17 |
| 4.3 | General Purpose GPU | 18 |
| 4.3.1 | CUDA | 18 |
| 4.3.2 | OpenCL | 18 |
| 5 | Parallel Computing using Nvidia CUDA | 20 |
| 5.1 | Parallel Computing | 20 |
| 5.1.1 | Theoretical Speedup | 21 |
| 5.2 | General Purpose GPU | 22 |
| 5.3 | Programming Model | 22 |
| 5.3.1 | Warps | 23 |

| | | |
|----------|--------------------------------------------------------|-----------|
| 5.3.2 | Blocks | 23 |
| 5.3.3 | Grids | 24 |
| 5.3.4 | Memory Hierarchy | 25 |
| 5.4 | Programming Guidelines and Challenges | 26 |
| 5.4.1 | Calling a GPU Kernel | 26 |
| 5.4.2 | Memory Allocation and Transfer | 27 |
| 5.4.3 | Implementation of a Simple Kernel | 28 |
| 5.4.4 | Shared Memory and Barrier Synchronization | 30 |
| 5.4.5 | Caveats and Lessons Learned | 32 |
| 6 | Tracking & Hough Line Transformations | 34 |
| 6.1 | Definitions | 34 |
| 6.2 | Algorithm | 36 |
| 6.3 | Peakfinding | 37 |
| 6.4 | Optimization | 40 |
| 6.4.1 | Paint by Numbers | 40 |
| 6.4.2 | Analytical Approach | 40 |
| 6.5 | Generalization to multi-D | 42 |
| 6.6 | Performance Comparison | 42 |
| 6.6.1 | Grid Dependence | 42 |
| 6.6.2 | Hit Dependence | 43 |
| 7 | Adjacency Graphs for Vertex Finding | 45 |
| 7.1 | Introduction | 45 |
| 7.2 | Converting an Event to a Matrix | 45 |
| 7.3 | Visualizing Events using Graphs | 46 |
| 7.4 | Construction | 47 |
| 7.4.1 | Accounting for Statistical Error | 49 |
| 8 | Adjacency Matrix Implementation and Performance | 50 |
| 8.1 | CPU | 51 |
| 8.2 | GPU | 52 |
| 8.2.1 | Memory | 53 |
| 8.3 | Optimization | 53 |
| 8.3.1 | CPU | 53 |
| 8.3.2 | GPU | 53 |
| 8.3.3 | Memory | 55 |
| 8.4 | Benchmarking | 55 |
| 8.4.1 | Caveat | 57 |

| | | |
|-----------|--------------------------------------------|-----------|
| 9 | Histogramming in Parallel | 59 |
| 9.1 | Introduction | 59 |
| 9.2 | CPU Histogramming | 59 |
| 9.3 | GPU Histogramming | 60 |
| 9.3.1 | Bin-Parallel Algorithms | 60 |
| 9.3.2 | Data-Parallel Algorithms | 61 |
| 9.3.3 | Parallel Implementations | 61 |
| 9.4 | Results | 63 |
| 10 | Primary Vertices from Histograms | 68 |
| 10.1 | Optimal Bin-Width | 68 |
| 10.1.1 | Results | 69 |
| 11 | Secondary Vertices | 73 |
| 11.1 | Candidates | 73 |
| 11.2 | Origin Cut | 73 |
| 11.2.1 | Beaming | 75 |
| 11.3 | Radial Cut | 77 |
| 11.3.1 | Cylinder Cut | 78 |
| 11.4 | Connected Components | 78 |
| 11.5 | Example | 79 |
| 11.5.1 | Origin Cut | 80 |
| 11.5.2 | Radial Cut | 80 |
| 11.5.3 | Connected Components | 81 |
| 11.6 | Results | 81 |
| 12 | Conclusion | 83 |
| 12.1 | Histogramming | 83 |
| 12.2 | Graph Cuts | 84 |
| 12.3 | Outlook | 84 |
| A | Histogram Implementations | 85 |
| A.1 | Data-Parallel: Naive | 85 |
| A.2 | Data Parallel: Shared Memory | 85 |
| A.3 | Bin Parallel: Naive | 86 |
| A.4 | Bin Parallel: Blocks | 87 |
| A.5 | Bin Parallel: Warps | 88 |
| A.6 | Bin Parallel: Warps + Blocks | 88 |
| A.7 | Bin/Data Parallel: Shared Memory | 89 |

Chapter 1

Introduction

One of the most urgent questions in modern physics, and to humanity in general, is how the universe came to be. More specifically, how the universe could have evolved to what it is at present day. Many theoretical and experimental efforts have been made to pursue the answer to these questions, but the Large Hadron Collider might be the most astounding result of this quest today.

This 27-kilometer-long ring of superconducting magnets aims to collide protons at incredibly high energies, in order to study the ‘stuff’ that results from these collisions (Figure 1.1). LHCb

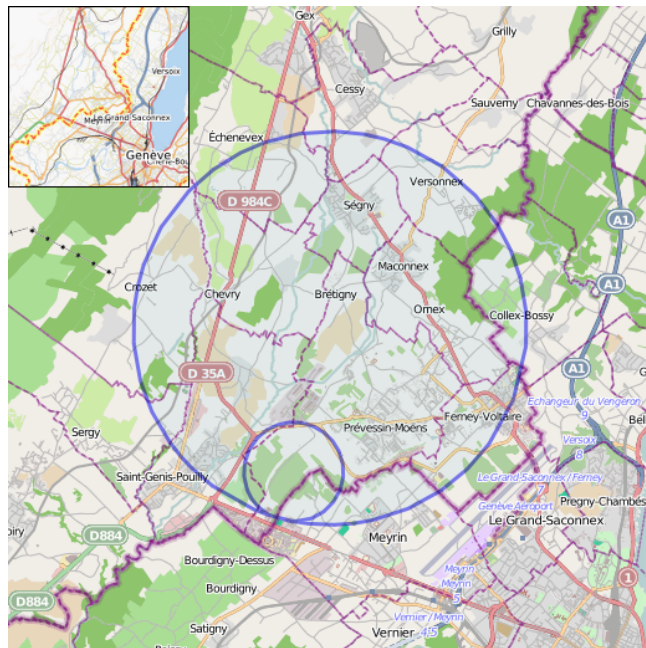


Figure 1.1: Location of the LHC near Geneva, Switzerland (image courtesy of Wikipedia, licenced under the Creative Commons (CC BY-SA) licence: <http://creativecommons.org/licenses/by-sa/2.0>)

is one of four detectors (Atlas, Alice and CMS being the other three) in which such collisions take place. Its particular goal is to measure properties of B-mesons, mesons that contain a bottom antiquark \bar{b} . An important part of the data-processing is the *trigger*, which decides whether the data from a collision-event is stored to disk for later analysis, or not. Because events occur at ~ 40 MHz, this trigger has to operate fast and efficiently.

With the increase of event-rates due to even higher energies after future upgrades in mind, the feasibility of new hardware to speed up the process is being investigated more and more. The development of mobile processors (ARM architecture) has spawned a sudden and explosive improvement on energy-efficient computing power that might be useful for this purpose. The processing cores might be slower than conventional CPU's, but they are also much smaller and less energy consuming. Field Programmable Gate Arrays (FPGA) might be used to process data efficiently, because they are aimed at executing one application only, programmed effectively in hardware. Last but not least, Graphics Processing Units (GPU) can be exploited to perform calculations with an inherently parallel character.

This report will focus on the design and implementation of a proof-of-concept trigger to run partly on a General Purpose GPU (GPGPU). We will be using the NVIDIA CUDA platform to investigate how practical and feasible it is to use GPU's as additional workhorses in the trigger. However, as the current software has been designed for serial devices, its algorithms are very sequential in nature. Therefore, this report will explore new possibilities and approaches to the problem, this time with parallelism in mind.

Chapter 2

LHCb Detector

2.1 Introduction

The LHCb experiment is one of the four large experiments at the Large Hadron Collider (LHC) at CERN (Geneva). Its detector is a forward spectrometer, suitable for performing precision measurements. The particles of interest have relatively low transverse momenta, and therefore travel along the beamline. This is due to the momentum distribution of partons (quarks and gluons) within the colliding protons. Heavy particles such as B mesons of interest tend to form out of collisions between a high-momentum quark on one side, and a low-momentum quark on the other. The resulting particle therefore is boosted in the direction of the high-momentum quark. The detector geometry (Figure 2.2) reflects this, being oriented away from the collision point (which is inside the VELO, located on the far left of Figure 2.2). Each of the components serves a very specific purpose, and will be briefly touched upon in the coming sections.

2.2 Vertex Locator (VELO)

The proton beams meet and collide inside the Vertex Locator (VELO), the goal of which is to reconstruct the geometry and physics of this collision. The VELO consists of radially and axially segmented silicon detectors that surround the beam axis and record *hits* when a charged particle travels through them. These hits are then combined in a pattern-recognition algorithm to form tracks, the paths along which the particles have traveled from their point of origin. Once the tracks have been determined, the decay topology is reconstructed based on common points of origin of different tracks. These points are called *vertices*, and can be partitioned into primary and higher order, mostly secondary, vertices.

The primary vertices are the coordinates of collision, i.e. the locations (on the beam-axis) where the pp-collision took place. The vast majority of tracks originate from these primary vertices (PV), and for this reason they can be reconstructed relatively easily. The secondary vertices (SV) however, are displaced (typically in the order of millimeters for B 's) from the PV and form the point of origin of only a few tracks. These SV's correspond to the decaypoint of

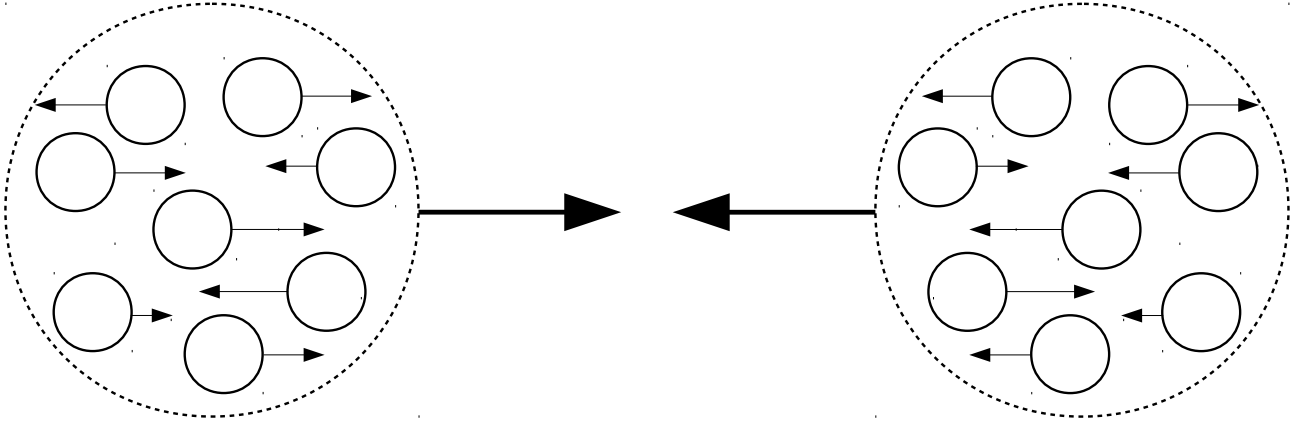


Figure 2.1: Simplified distribution of partons (quarks and gluons) within a nucleon. Colliding partons with highly differing relative momenta that collide, will produce new particles that are boosted in either direction.

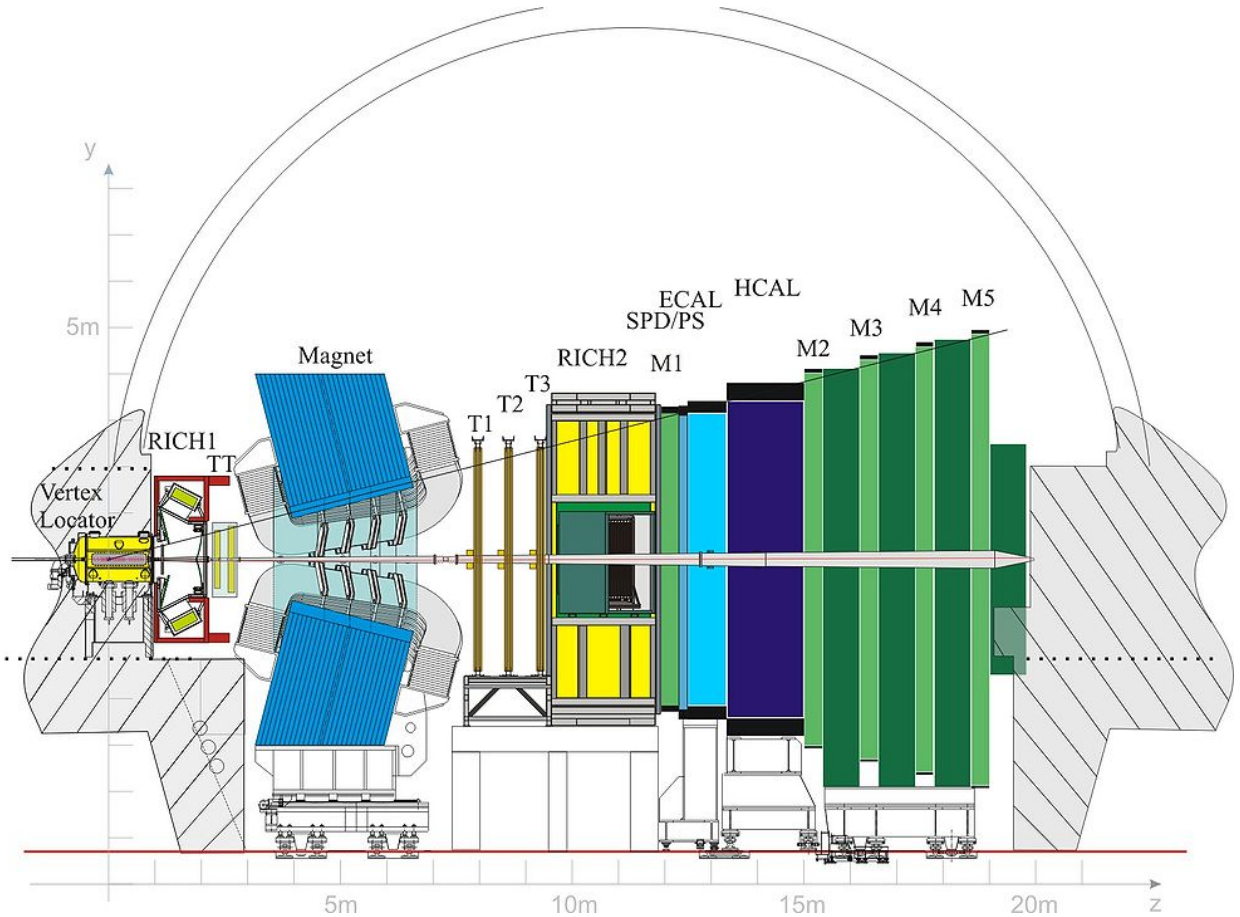


Figure 2.2: Cross-section of the LHCb detector (image from [3]).

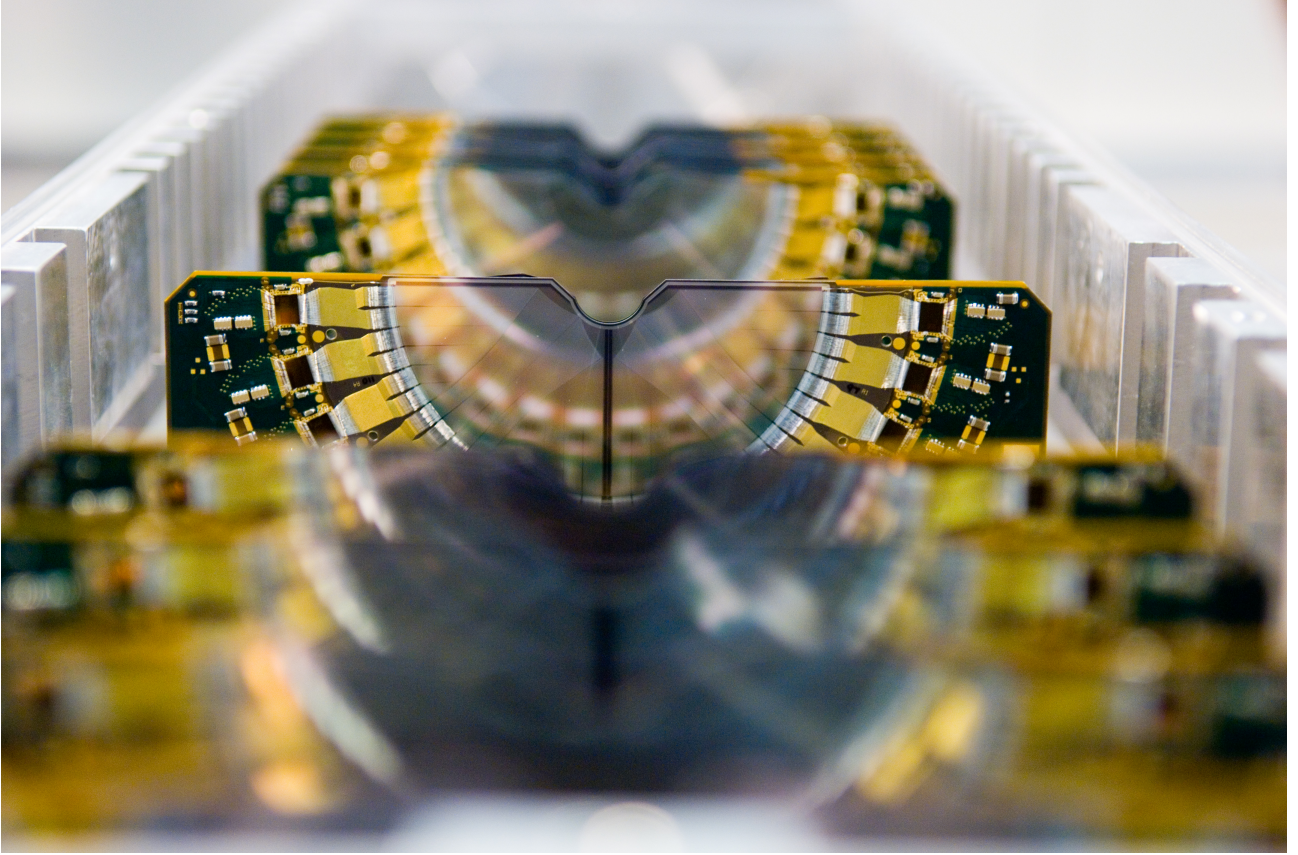


Figure 2.3: The VELO silicon detector strips (Source: CERN). The beamline goes ‘into’ the picture, through the semi-circular cut-outs in each of the 21 strips. The aluminium RF-box encloses the array of strips, separating them from the LHC vacuum.

short-lived particles, and in particular at LHCb: B-mesons. Because of the importance of the VELO in this project, more details will be given in Section 3.

2.3 RICH 1 and 2

The Ring Imaging Cherenkov detectors come in a pair to provide particle identification (PID), which is essential information for the reconstruction process. The first detector (RICH1) is located upstream of the magnet, directly after the VELO and covers the low-momentum range ($\sim 2 - 50$ GeV). The second (RICH2) covers the high-momentum range (up to ~ 100 GeV), downstream of the magnet.

The detectors use the Cherenkov-effect [REF] to measure the velocity of the charged particle, which can in turn be used to determine its mass (and therefore its identity) when the momentum is known from the deflection in the magnet. The Cherenkov effect occurs when a particle travels through a medium faster than the local speed of light, emitting a cone of light (often compared

to the sonic boom). The angle at which this cone is emitted depends on the velocity, according to

$$\cos \theta = \frac{1}{n\beta} \quad (2.1)$$

where n is the medium's index of refraction and $\beta = v/c$ the particle's velocity relative to the speed of light. This cone is generated in a thin layer of aerogel ($n = 1.03$) [1] after which it is projected onto the photodetectors through two mirrors to focus and extend the pathlength.

Measuring the radius of the ring of light is a measure of the angle and therefore the particle's velocity.

2.4 Magnet

The magnetic field induced by the magnet bends the charged particles in a direction that depends on both the magnitude and sign of their charge:

$$\rho = \frac{p}{cqB} \quad (2.2)$$

where ρ is the bending-radius, p , q and c are the particle's (relativistic) momentum and charge, and B is the magnetic field strength. Measuring this curvature ρ allows for a measurement of momentum which, together with the velocity-information gained from the RICH-detectors, determines the particle-mass and therefore identity. Moreover, the magnet is the only part of the detector that can establish the *sign* of the particle's charge, since opposite charge will cause the particle to bend in the opposite direction.

2.5 Calorimeter

Downstream of the second RICH detector are located the calorimeters, used to trigger on electrons, photons and hadrons (including the neutral ones), and to provide energy-measures. The system consists of a scintillating pad detector (SPD), a Pre-Shower detector (PS), an Electromagnetic Calorimeter (ECAL) and a Hadronic Calorimeter (HCAL).

The PS is used to distinguish electrons and photons from charged hadrons. A layer of scintillating pads is placed behind a lead absorber that is too thin for charged hadrons to induce particle showers. Therefore, electrons and photons can be identified by the hardware triggers as significant energy deposits in both the PS and ECAL. To further distinguish between electrons and photons, another layer of scintillating pads is installed even before the lead-layer of the PS (SPD). Electrons travelling through this layer are likely to produce a signal, in contrast to photons.

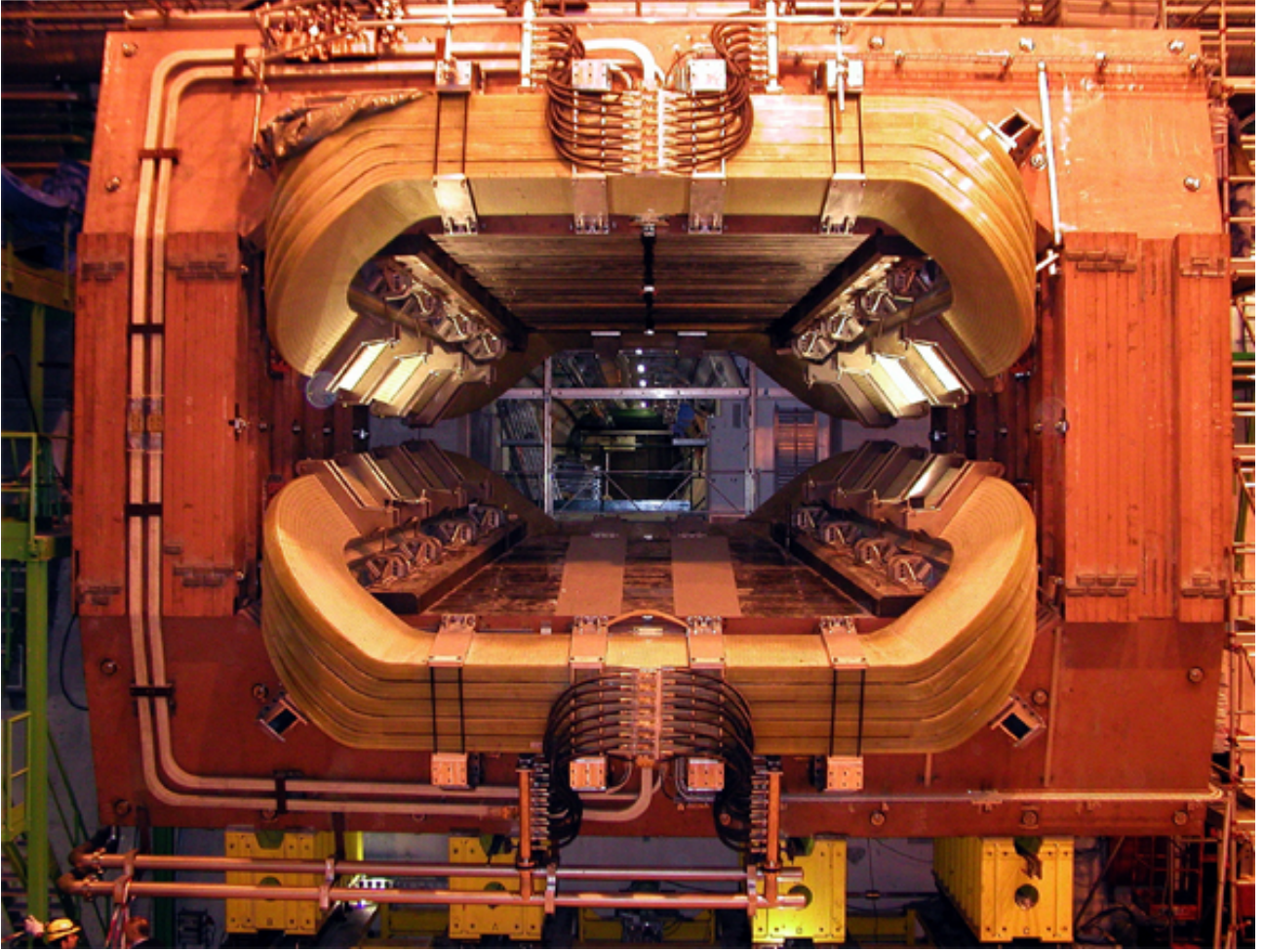


Figure 2.4: The LHCb magnet right after installation. (Source: CERN)

2.6 Muon Detector

The muon detectors are located at the far end of the detectors, partly in front of, but mostly behind the calorimeter system. They have been designed to provide information to the high p_T muon trigger both during online and offline analysis. Because generally, only muons are energetic enough to reach the outer muon station, it can be assumed that energy deposits in the muon detector are from muons only.

Chapter 3

Vertex Location

This research has been primarily focussed on the detection of primary and secondary vertices of short living particles. Therefore, this chapter is dedicated to the VELO and the methods employed to reconstruct the positions of such vertices inside the VELO.

3.1 Technology

Figure 3.1 shows a schematic side-view of the current (up to 2018) VELO, from which can be seen that each half consists of 21 two-sided silicon strips. The different sides contain R - and ϕ -sensors respectively to measure both the radial and azimuthal component of the particle-trajectory. Each of the strips therefore records *hits*, which are defined as an (R, ϕ) pair at the z -coordinate corresponding to a particular sensor-strip. To achieve maximum resolution, these detectors have to be very close to the interaction-point, and are therefore mounted at a radial distance of only 7mm of the beam. During the injection phase, when the beam is still much wider, the halves can be retracted to 30mm. To protect the LHC's primary vacuum (10^{-9} mbar) from outgassing of detector modules, a secondary vacuum (10^{-6} mbar) is established, separated from the primary vacuum by aluminium RF-boxes that also shield the modules against RF-pickup (Figure 3.2).

Due to the constraint that a track must have *at least* 3 hits in the VELO, tracks can only be detected up to a certain angle which depends on the geometry of the silicon strips. This is reflected in Figure 3.1, which shows the maximum angle of 390 mrad of a track that still passes through 3 strips.

A second limitation to the angular acceptance is the *hole*, i.e. the void through which the beam passes. Obviously, tracks that pass only through the void (at angles below 15 mrad) cannot be detected.

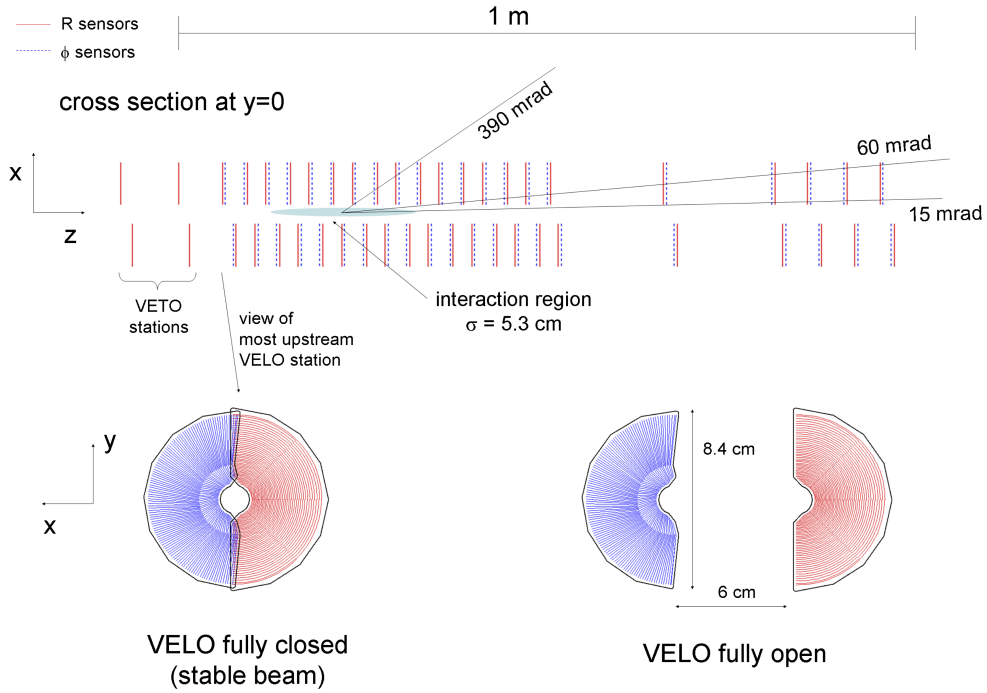


Figure 3.1: Schematic view of the Vertex Locator (VELO) in the xz - and xy -planes. The particle-beam runs along the z -axis. The semicircular silicon-strips are shown in the fully closed (7 mm from the beam) and fully open position (30 mm from the beam). Around the interaction region, the strips are positioned more densely to be able to record tracks with higher transverse momentum, that will leave the detector without passing through subsequent stages.

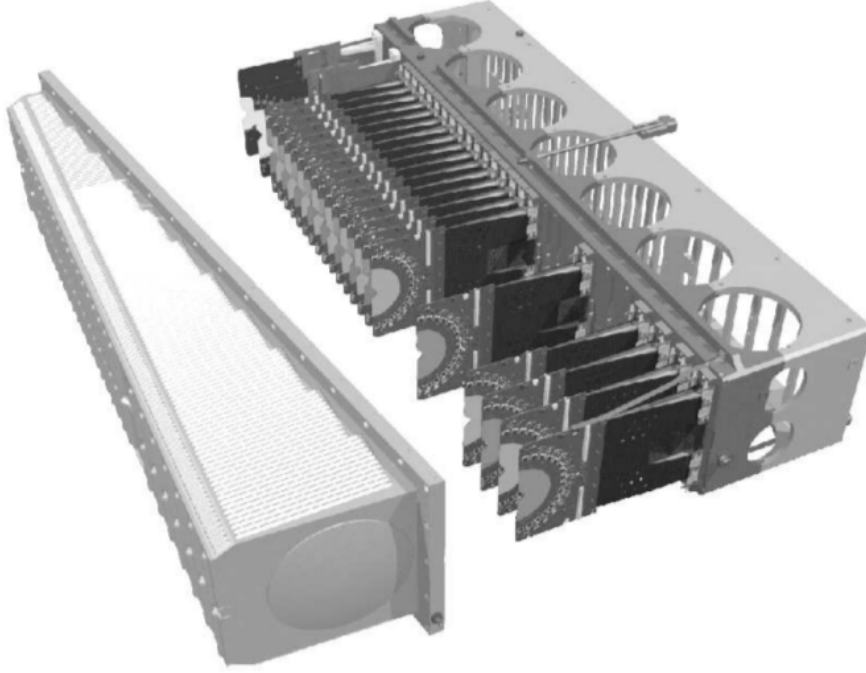


Figure 3.2: Silicon detector modules (21) mounted on their supports inside the VELO, and the surrounding aluminium RF-box (left).

3.2 Tracking

To find the exact coordinates of the decay-points, or vertices, the particle-tracks have to be reconstructed. This process is called tracking, and is primarily a combinatorial problem, where every possible combination of hits (in different layers) might represent a track. To reduce the computational complexity, a Kalman filter-based algorithm has been implemented [2] to find sets of hits that form tracks. A Kalman filter characterizes itself by iteratively (or recursively) making predictions, and correcting these predictions based on additional measurements. In the context of tracking, the algorithm will hypothesize a track by selecting two subsequent hits, which by definition will define a straight line through the remaining layers. It will then look for a hit in the subsequent layer, closest to the predicted line, and correct the track-parameters based on this new information (Figure 3.3). When the deviations become too large, the track is discarded. The resulting tracks are then passed on the vertex-finding algorithms.

3.3 Primary Vertices

Primary vertices are points in 3D-space where proton-proton interactions have occurred. During online analysis (while the LHC is running), these can be reconstructed with precision close to that of the offline analysis. Currently reconstruction of the vertices is a two-stage process,

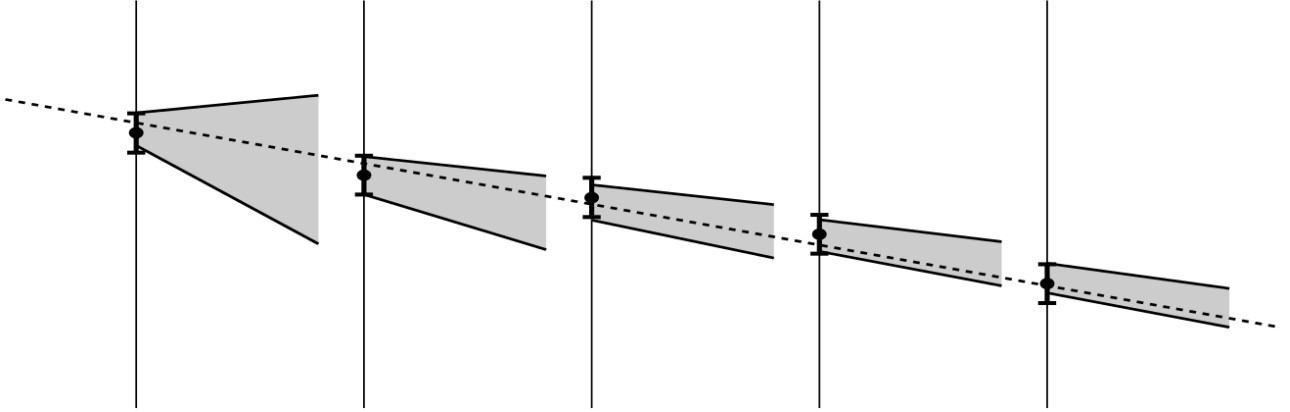


Figure 3.3: Visualization of the Kalman-filter based tracking algorithm, applied from left to right [2]. The vertical lines and corresponding dots represent the detector planes and recorded hits and errors. The cones represent the uncertainty in the track parameters, which get smaller with each step.

consisting of a *seeding* and *fitting* phase [7].

3.3.1 Seeding

For online reconstruction the *fast seeding* method is used. This is a clustering algorithm that tries to find clusters of tracks along the beam line, i.e. tracks with similar points of closest approach with the z -axis. Each cluster is defined by its coordinate z^{clu} and σ_z^{clu} , where σ_z^{clu} depends on the angle θ of the track with respect to the beam line.

In each iteration, the pair of tracks with minimum distance $|z^{\text{clu}_1} - z^{\text{clu}_2}|$ is selected, and the distance parameter D_{pair} is defined as

$$D_{\text{pair}} = \frac{|z^{\text{clu}_1} - z^{\text{clu}_2}|}{\sqrt{(\sigma_z^{\text{clu}_1})^2 + (\sigma_z^{\text{clu}_2})^2}}. \quad (3.1)$$

If $D_{\text{pair}} < 5$, the clusters are merged and the new cluster-parameters are calculated from the weighted means and variances:

$$z^{\text{clu}_3} = \frac{(\sigma_z^{\text{clu}_1})^2 z^{\text{clu}_1} + (\sigma_z^{\text{clu}_2})^2 z^{\text{clu}_2}}{(\sigma_z^{\text{clu}_1})^2 + (\sigma_z^{\text{clu}_2})^2} \quad (3.2)$$

$$(\sigma_z^{\text{clu}_3})^2 = \left[\left(\frac{1}{\sigma_z^{\text{clu}_1}} \right)^2 + \left(\frac{1}{\sigma_z^{\text{clu}_2}} \right)^2 \right]^{-1} \quad (3.3)$$

The list of clusters is iterated as long as merging still takes place. When terminated, the resulting clusters and their parameters are used to seed the next stage: fitting.

3.3.2 Fitting

The method described in Section 3.3.1 only yields suboptimal rough estimates of the primary vertex positions. At the time of reconstruction, it is still unknown which tracks correspond to decay products. Therefore, tracks originating at secondary vertices are still likely assigned to a PV, systematically biasing the estimated z -coordinate towards the SV. Moreover, badly measured tracks like ghost tracks or multiply scattered tracks decrease the resolution of the PV position. The fitting-stage was developed to counter these effects, using an adaptive weighted least squares method.

This method implements an iterative procedure in which the estimated z -coordinate is refined in each iteration by minimizing the statistic

$$\chi_{\text{PV}}^2 = \sum_{i=1}^{n_{\text{tracks}}} W_{T,i} \cdot \chi_{\text{IP},i}^2, \quad (3.4)$$

where W_T is the Tukey weight. This weight-factor is defined such that tracks with a high χ_{IP}^2 are excluded from the set of tracks assigned to this PV:

$$\begin{aligned} W_T &= \left(1 - \frac{\chi_{\text{IP}}^2}{C_T^2}\right)^2 & \text{if } \chi_{\text{IP}} < C_T \\ W_T &= 0 & \text{if } \chi_{\text{IP}} \geq C_T. \end{aligned} \quad (3.5)$$

The Tukey constant $3 \leq C_T \leq 10$ is a measure that determines the threshold of selected tracks. Each iteration consists of the following steps:

1. Given some C_T , calculate χ_{IP}^2 and $W_T(\chi_{\text{IP}}^2)$ for each of the tracks.
2. Minimize Equation 3.4 with respect to z_{IP} , while keeping W_T constant (even though this is strictly a function of χ_{IP}^2 which in turn is a function of z_{IP}).
3. Determine the new value of C_T and repeat from (1), using the new value of z_{IP} to calculate the weights.

The iteration ends when $|\Delta z| < 0.5\text{mm}$, where Δz denotes the shift of the z -coordinate w.r.t. the previous iteration, provided that at least 5 tracks have been assigned to the PV. Otherwise, a maximum of 30 iterations is performed.

3.4 Secondary Vertices

When looking for B -mesons, only events containing secondary vertices are deemed interesting enough for storage on disk. However, total reconstruction of such events is too time-consuming for online analysis. The trigger therefore looks for a simple and fast tell-tale signal of secondary vertices: a large impact parameter with respect to the PV's.

Given a set of PV coordinates along the beam-line (z -axis), the impact parameters of tracks w.r.t these points are calculated. Events containing tracks with a minimum IP larger than some threshold are selected as candidates and stored for offline analysis [12].

Chapter 4

Motivation

4.1 Current Affairs

When the LHC is running at full capacity, producing 40 million events per second in the LHCb detector, it is impossible to store each event for later analysis. This would simply take up too much (non-volatile) memory. The trigger system was designed to reduce this rate to a more manageable level, such that seemingly interesting events can be sent to disk in anticipation of later (offline) processing. The hardware and algorithms involved in this triggering process are therefore highly constrained by timing requirements.

The triggering procedure used during run 1 and run 2 (pre Long Shutdown 1, 2012) can be subdivided into two phases: the low-level L0 and high-level trigger (HLT). The L0 trigger is implemented in hardware and reduces the rate to about 1 MHz before it is passed on to the HLT, which is a full software implementation aimed at a further reduction to about 5 kHz. The implementation of the HLT is called Moore, and is being run in a computing farm consisting of over 27,000 (physical) cores [4]. This means that each processing unit running Moore has a window of about 20 ms to process an event in order to determine whether or not to keep this event for future analysis.

4.2 Upgrade

After *Long Shutdown 2* (2018-2019), both the intensity and energy of the LHC beam will be increased. Consequently, there will be events at a higher rate, containing more proton-proton interactions. Several improvements will be made, both in hardware and in software, to compensate for this increase. Currently, the hardware-implemented L0 trigger is throttled by the HLT to make sure the rate is manageable. After the upgrade, the L0 should be effectively eliminated from the trigger pipeline, being replaced by the Low Level Trigger (LLT), implemented entirely in software. By having the entire system implemented in software, a more flexible system is established.

Another improvement is the implementation of the *deferred trigger* system. This is basically

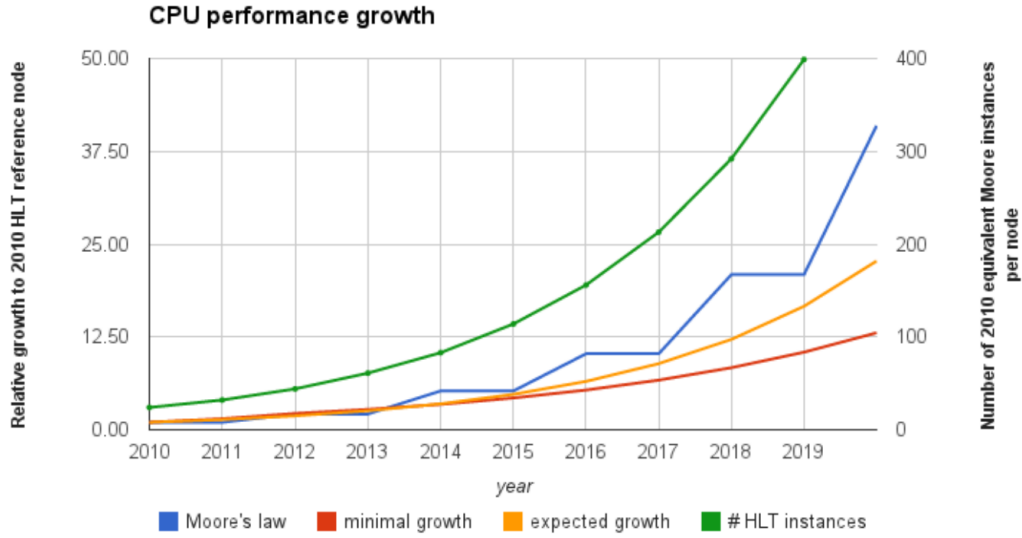


Figure 4.1: Relative expected CPU growth with respect to a 2010 reference node [8]. After LS2 (2019), a 16 fold increase in CPU power can be expected from Moore’s law, which would result in 50 instances of Moore (the HLT implementation) being run on each CPU node.

a cache system, allowing roughly 20% of all L0 accepted (LLT in the future) events to be stored directly on disk without passing through HLT [6]. Processing of these events is deferred to the idle-time during the LHC re-fill. This technique allows the current L0 trigger to accept events on a looser p_T threshold (500MeV/c to 300MeV/c), reducing the risk of omitting interesting events.

4.2.1 Future Performance

Keeping in mind the upgrades in computing hardware and tracking technologies like the VELO Pixel detector (replacing the current silicon detector modules), it is estimated that the HLT can be completed in less than 10 ms [8], where the maximum processing time is estimated at 13 ms (Figure 4.1).

These numbers are based on the current algorithms and selection procedures. Although it is foreseen that it will suffice to run the current selection algorithms within an acceptable time-window, there will not be much room for more sophisticated triggering algorithms to reduce the rate even more in the future. When the intensity is increased, it will be more and more likely that the current selection criteria will yield interesting events. It is therefore desirable to have a trigger system that can select events on more stringent criteria, i.e. a system that performs a larger part of the analysis that is normally done only offline. The most obvious augmentation of the analysis is inclusion of SV detection. To accomplish this, additional hardware is needed to function alongside the next generation of CPU nodes in the farm. The remainder of this section will introduce the candidate architecture of our choice: Graphics Processing Units (GPU’s).

4.3 General Purpose GPU

Because Moore's law alone will not be enough to compensate for our wish to find SV's in the trigger, it is interesting to investigate different architectures like GPU's that may extend the computing capabilities of the farm. As the name suggests, GPU's have traditionally been used to calculate graphics in both professional and consumer applications. When gaming and CGI (Computer Generated Images) became more and more popular, the competitive market in graphics cards spawned many generations of increasingly fast graphics processors for relatively low prices.

A more recent development is that these cards are increasingly being used in other types of applications. They are highly optimized for massively parallel tasks, like calculating the color of every individual pixel on a computer screen. Therefore, any calculation that features many independent and similar calculations can be done with high efficiency on these parallel architectures, which has led to the development of the General Purpose GPU (GPGPU). Many voices in the tech world have uttered that Moore's law has come, or will soon come, to an end when CPU power it concerned. Although this is hard to confirm or disprove given the wildly varying statements on the topic, it is certain that the GPU industry is still growing stronger and chips are getting faster because more and more parallel cores can be embedded in the same module.

This increasingly parallel nature of GPU's naturally invokes the need to rethink algorithms to exploit this architecture, which is where the challenge lies. This research will make an effort to opening the gates of rethinking vertex-location in a parallel fashion that comes natural to the GPU architecture.

4.3.1 CUDA

Nvidia, a company specialized in the production of GPU technology, has seen the merit in this technology and designed its more recent generations of hardware to reflect the general purpose use of its products. The CUDA (Compute Unified Device Architecture) framework is a parallel programming model and platform that provides developers direct access to the instruction set and memory of supported cards (Geforce 8-series and beyond). Extensions of mainstream programming languages like C++ and Python allow developers access to the hardware with a relatively low threshold and moderate learning curve. This low threshold combined with the vast amounts of documentation, its well established reputation and the highly active developmental status have been the reasons to adopt CUDA as the platform of choice for this project. Section 5 will provide more information on the CUDA architecture.

4.3.2 OpenCL

A viable alternative to CUDA would have been OpenCL. This is a more general platform that allows not only GPU's, but many different processing units contained in the system. Like CUDA, it provides a C-like language and API for addressing this hardware. The great advantage

of OpenCL over CUDA is that it can be run on *any* GPU, not only the ones supported by Nvidia. The code will even run when a supported GPU is not available in the system, because it can run on the CPU as well. This allows for writing programs that are very flexible, taking advantages of the resources available at runtime. However, for specific purposes like ours, when it can be known in advance what the hardware specifications will be, this advantage vanishes and CUDA seems to be a better fit to the needs.

Chapter 5

Parallel Computing using Nvidia CUDA

5.1 Parallel Computing

The quintessential notion of parallel computation is that multiple calculations are done at the same time by separate computing units. There are many different architectures that support this, at various levels. *Bit-level* parallelism is the simplest and oldest form, where the size of *words*, memory-chunks that can be processed by a single instruction, was increased to be able to process larger numbers in a shorter amount of time. Early digital computers were based on 4-bit CPU's, which meant that these computers could only operate directly on 4 bits at the time. When numbers outside the range $[0, 15]$ (unsigned) or $[-7, 7]$ (signed) needed to be processed, multiple words had to be processed separately and their results combined to get the final result. Today, most computers have 64-bit processors, capable of dealing with numbers large enough for almost every field of application¹.

In *instruction-level* parallelism, independent instructions are identified and scheduled such that the total throughput is maximized. Even single-core CPU's perform multiple actions at the same time, i.e. the clock pulse is directed to multiple places. Memory reads/writes, instruction fetches, decoding and execution can all be performed simultaneously. Instruction pipelining is a way of maximizing throughput by reordering instructions to hide latency. A 4 stage-pipeline is shown in Table 5.1, which allows the CPU to perform 4 subsequent instruction executions (EI) in a row, instead of having to wait until the next instruction and operands have fetched and decoded.

Task-parallelism is accomplished by assigning different tasks, or processes, to different CPU cores (threads). These processes can be entirely different from each other, operating on entirely different data. A much seen example of this is the household desktop PC, running an operating

¹In the field of encryption, where for example very large primes are concerned, the algorithms work on numbers that exceed the 64 bits available to the CPU. Here, the software still relies on libraries that perform these calculations in software rather than hardware.

| instr. | cycle | | | | | | | | |
|--------|-------|----|----|----|----|-----------|-----------|-----------|-----------|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 6 | 7 |
| 1 | | FI | DI | CO | FO | EI | WO | | |
| 2 | | | FI | DI | CO | FO | EI | WO | |
| 3 | | | | FI | DI | CO | FO | EI | WO |
| 4 | | | | | FI | DI | CO | FO | EI |

Table 5.1: Example of a 4-stage pipeline where the operations FI (Fetch Instruction), DI (Decode Instruction), CO (Compute Operand Address), FO (Fetch Operand), EI (Execute Instruction) and WI (Write Operand) can be executed in parallel. Due to this being a 4-stage pipeline, it is possible to execute 4 instructions (EI) in the course of 4 clock-cycles.

system and various applications. These applications might be executing in parallel on different cores if the CPU is a multi-core processor. This is in sharp contrast to *data-parallelism*, where a single process tries to speed up its execution by distributing workload across threads. In contrast to task-parallel processes, these threads operate on the same data and need to communicate to each other, whereas parallel tasks need not know anything about one another. Moreover, the author of data-parallel programs has to think about his algorithms in order to parallelize them, whereas it is the operating system that takes care of task-parallel processes.

5.1.1 Theoretical Speedup

Given a total of p processors, an algorithm can theoretically be sped up by a factor p (ignoring the possibility of *super-linear* speedup due to e.g. cache effects). However, in practise not every bit of an algorithm is suitable for parallization. Therefore, the maximum theoretical speedup was formulated by Gene Amdahl as a function of the fraction of time α which a program spends in non-parallelizable parts:

$$S_{\alpha}(p) = \frac{1}{\frac{1-\alpha}{p} + \alpha}, \quad (5.1)$$

which, in the limit of $p \rightarrow \infty$, converges to $1/\alpha$. Equation 5.1 is known as Amdahl's Law, which basically means that regardless of the number of processors added to the system, the maximum speedup is constrained by the amount of parallelizable portions in an algorithm.

In the context of General Purpose GPU's (Section 5.2), there is often the need to compare GPU performance across many cores (large p) to the performance of a single or a few CPU cores. Before this can be done using Amdahl's law, one has to realize that a single GPU core is much slower, partly because it runs at a much lower clock frequency and has smaller caches, than a single CPU core. If we set a single CPU-core equal to β GPU cores, Amdahl's law becomes

$$S_{\alpha}^{\beta}(p) = \frac{1}{\frac{1-\alpha}{\beta p} + \alpha}, \quad (5.2)$$

where $\beta < 1$. Equation 5.2 still converges to $1/\alpha$, though at a smaller rate due to the additional factor β . The above is still excluding overhead due to data transfer, which will be discussed in the upcoming chapters.

5.2 General Purpose GPU

It was not until the late 1990's that the concept of a Graphics Processing Unit first emerged in the form of the Nvidia RIVA 128 and later the Nvidia Geforce Fx. Prior to this, only Video Graphics Array (VGA) controllers existed to accelerate 2D graphics in user interfaces. With consumer gaming becoming more and more popular, these devices started to evolve from simple configurable graphics processors to complex and freely programmable parallel computing platforms. At the moment of writing, the GPU is the most pervasive massively parallel platform available.

Their development has been mainly driven by the need for high performance real-time graphics rendering in games at relatively low cost to be available to consumers. In contrast to the development in CPU technology, which for a very long time was still mainly focussed on maximizing clock-speed, the GPU industry sought ways of maximizing parallelism, given the massively parallel nature of the process of rendering. Game developers want to write single programs, known as kernels or shaders depending on the context, that calculate the color of a single pixel as a function of its position on the screen. Given the vast amount of pixels on a screen, this same kernel has to be processed many times. GPU's enable many such kernels to be calculated in parallel, boosting performance.

When GPU's became more flexible, developers started to realize that this parallel design could also map to problems outside the realm of real-time graphics. Rendering API's were then 'abused' to solve completely different tasks, taking advantage of their ability to use many parallel cores separate from the CPU. This however was a complex matter programmatically, as it required molding the original problem into one that could be represented by graphical operations.

Then, in 2006, with the advent of the Geforce 8800, it became possible to write General Purpose GPU programs without the need to translate the problem to a graphical problem first. Nvidia introduced the Compute Unified Device Architecture (CUDA), which made it possible to execute arbitrary code in parallel on the GPU, in addition to DirectX and OpenGL. It uses the familiar C programming language, only marginally extended with some extra syntax and keywords. The next sections will cover these language extensions and how they represent the underlying CUDA architecture.

5.3 Programming Model

In any parallel computing environment, *threads* represent streams of instructions that are processed independently from each other. In the CUDA programming model, these threads are

organised in a hierarchy consisting of *warps*, *blocks*, and *grids*. Warps (Section 5.3.1) consist of 32 threads that execute in lock-step. Warps are then grouped into blocks (Section 5.3.2). Finally, blocks are organised into a grid, each of which still executes the same kernel. However, depending on the number of blocks and the number of threads within each block, the order of execution is undefined.

5.3.1 Warps

As mentioned before, warps execute in lock-step. This means that within a warp, threads are all given the same sequence of instructions, even if part of them do not have to execute this execution (due to branching, Figure 5.1). The programmer has full control over the number of (logical) threads within a block, but the CUDA runtime will always assign a multiple of 32 threads to the block, as warps can not be split across block-boundaries. Each thread ‘knows’ whether or not is currently active and should execute the instruction or not. Non-active threads are therefore simply waiting until their branch becomes active again. Consequently, there is never any need for barrier synchronization between threads in a warp, which is useful when data has to be shared between such threads. Furthermore, CUDA provides so-called voting functions that allow for very efficient communication (and data-sharing as of Compute Capability 3.5) between intra-warp threads in some common situations. This will be discussed and used in more detail in Section 9.3.3.

While the warp-oriented design can be used to the programmers advantage, the danger of branching should be taken very seriously when writing a kernel. Often, code can be organised to minimize the number of divergent paths within warps, which could have serious performance impact.

5.3.2 Blocks

A block is a selection of threads that is executed on the same Streaming Multiprocessor (SM). A SM is a physical part of the GPU that contains multiple CUDA-cores, each of which is able to execute multiple threads. The number of SM’s and the number of CUDA-cores within each SM is device-dependent and indicated by the so-called *compute capability* of the GPU.

Because of the fact that threads are always grouped in warps, each block contains a whole multiple of the warp-size (32) even when the programmer has provided a smaller number. In the latter case, the remaining threads of the warp is just idle for the entire duration of the kernel. Threads within a block all share a common patch of memory called *shared memory* which is local to the chip and therefore limited in size but very fast. It can be compared to the cache on a CPU, the major difference being that the programmer has full control over this cache. It is mainly used to share common data between threads, regardless of what warp these threads are in.

Threads within a block can be ordered in a 1, 2 or 3-dimensional array. The ability to represent threads in a multi-dimensional grid is obviously the legacy of GPU’s in rendering 2D and 3D scenes. The choice of block-dimensions depends solely on the particular problem and

how this is best represented in the abstract world of code. Each thread then has a thread-ID consisting of 1, 2 or 3 components depending on the dimensions of the block. For example, within a 3-dimensional block, each thread-ID has an x , y and z component, each of which can be queried inside the kernel.

5.3.3 Grids

Blocks are organised in a grid much like threads are organised in a block. A grid can be 1, 2 or 3 dimensional and each block has a block-ID containing as many components (x, y, z) as the number of dimensions of the grid. When threads in different blocks need to share data, this needs to be done through the *global memory*. This is the largest memory area (in the order of several Gigabytes for modern devices), but also the slowest to access. Again, this presents an area of optimization, and the need to design algorithms and implementations to minimize access to global memory and keep communication local to blocks such that shared memory can be exploited.

It is also possible to launch multiple kernels, and therefore multiple grids, asynchronously using streams. As long as the hardware permits it, grids running in different streams can run concurrently. This allows for example for manually managed pipelining of algorithms to fully saturate the GPU.

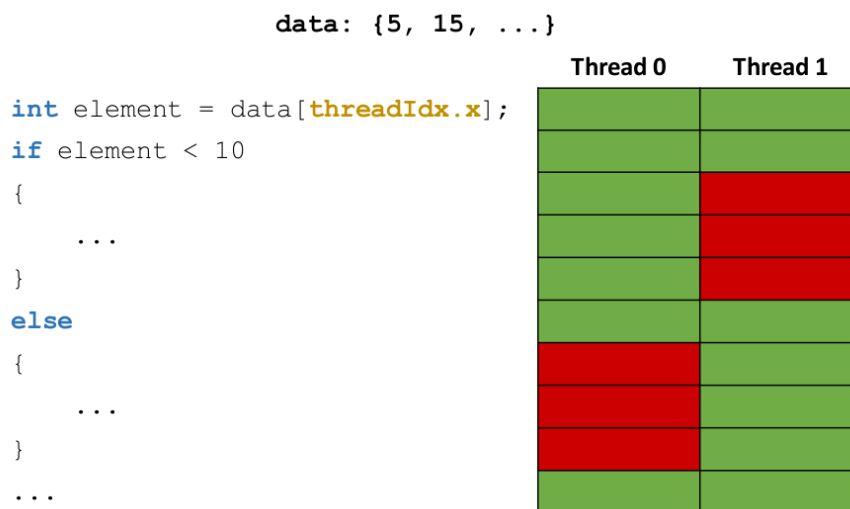


Figure 5.1: Illustration of idling threads due to branching within a warp. The instructions in the body of the if- and else-statements are executed depending on runtime decisions, the outcome of which may differ accross threads in a warp. Because these threads execute in lock-step, threads have to wait for their partners to finish before continuing execution.

5.3.4 Memory Hierarchy

The hierarchy described in previous sections is strongly correlated to the way memory is organised in CUDA. There are different kinds of memory, and the programmer has full control over which type of memory is used at which point in the application. Each of the different memory segments will be described below. A schematic overview is shown in Figure 5.2.

Thread-local Memory

Thread-local memory is memory accessible only to a single thread. When declaring a variable on the stack, these are either stored in registers or in off-chip (global) memory when not enough registers are available. As long as the variables are stored in local registers, thread-local memory is the fastest type of memory available and should therefore be used whenever applicable. The main drawback of using local memory is its limited size. Depending on the chip's compute-capability, each thread has access to only 63-255 registers (the exact specification can be found in the Cuda Programming Guide [11]). Because of this limitation and the fact that the compiler has no way to index arrays stored in registers by means of runtime indices, it is recommended to avoid storing arrays in thread-local memory.

Shared Memory

Shared memory is on-chip memory shared between all threads in a block, and therefore provides a very fast means of intra-block-communication. It can be used to manage a custom cache, provide fast access to data that is used by every thread in a block, or communicate data between threads. Because this memory is accessible to threads in different warps (which are not implicitly synchronized), one has to make sure race-conditions are avoided. This can be accomplished by calling the CUDA built-in function `--syncthreads()` to invoke a *barrier synchronization* for every thread *within a block*.

Global Memory

Global memory (also known as main memory or device memory) is available to every thread, and in very large quantities. This is at the expense of read and write latencies, which are high relative to previously mentioned types of memory. Writing code that takes cache locality into account is therefore critical when writing programs that make heavy use of this type of memory. Cache locality means that elements that are read subsequently from main memory are close to one another. When a value is read into a register, a cache line is filled containing data surrounding this value. Subsequent reads of this surrounding data can then be taken from the fast cache instead of having to go to the slower main memory again.

Constant Memory

Constant memory is a special case of device memory which is cached in the *constant cache*. The fact that the actual data in main memory is guaranteed to be identical to the data in

this cache, can be exploited during optimization. Storing variables in constant memory can therefore yield better performance.

Texture Memory

Texture memory is simply global memory where the way data is cached in a way that is optimized for 2D memory representations, in the *texture cache*. Instead of caching linearly, data surrounding the requested address in 2 dimensions is cached. This reflects the need of accessing data on a surface rather than a linear array, which is common in processing 2D graphics.

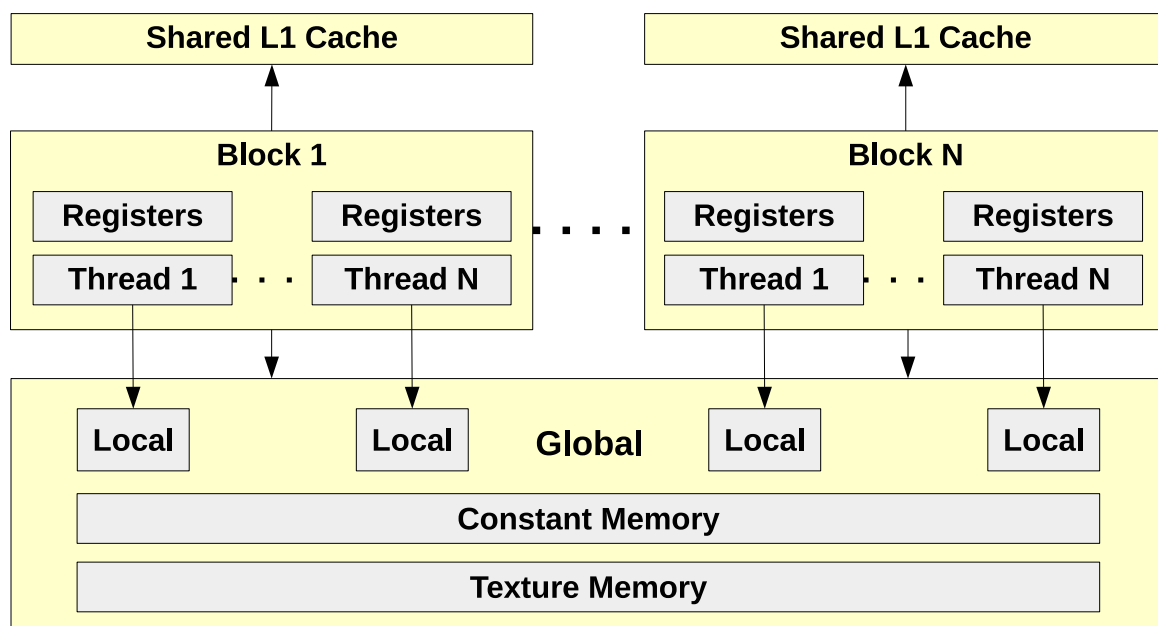


Figure 5.2: Memory hierarchy of CUDA, as described in Section 5.3.4

5.4 Programming Guidelines and Challenges

Given the programming model described in Section 5.3, the coming sections will show what these concepts look like in a code-editor, and what to watch out for when writing code and debugging.

5.4.1 Calling a GPU Kernel

A piece of code that executes on the GPU (also referred to as the *device*) is called a *kernel*. It is like a function that is called by the CPU (or *host*), transferring execution to potentially many threads on the GPU. These kernels never return a value (they have to return `void`), and

the CPU will not wait until a kernel returns. A kernel declaration looks just like an ordinary function declaration, except for the `--global--` modifier and obligatory void return type:

```
1 --global-- void kernel();
```

When calling this kernel from the host, some new syntax is encountered:

```
1 kernel<<< B, T >>>(); // calling the kernel  
2 // host will continue executing the code below a kernel-call
```

Here, *B* is the number of blocks (organized linearly within the grid) and *T* is the number of threads per block (organized linearly within the blocks). These parameters are known as *launch parameters*.

This mechanism alone is far from sufficient to take advantage of the GPU, because apart from passing value-parameters (`int`, `double`, etc) to the kernel, it not yet provides us with the means to transfer data between the host and device.

5.4.2 Memory Allocation and Transfer

Because the memory segments on the host and device are separated by the PCI-e bus, it is possible but highly inefficient for device-code to directly operate on host-memory². Instead, one should move the relevant data to the GPU memory where it can be processed accordingly. The CUDA API provides the means to manage memory on the device through functions like `cudaMalloc`, `cudaMemset` and `cudaMemcpy`. These functions are the equivalents of the familiar C standard library functions `malloc`, `memset` and `memcpy`, and allow the programmer to allocate and populate (global) memory on the GPU, and copy data back and forth respectively. This brings us to the canonical structure of a CUDA program:

1. Allocate sufficient memory on the host.
2. Allocate sufficient memory on the device (e.g. using `cudaMalloc`).
3. Optionally initialize the memory using `cudaMemset` or a dedicated initialization kernel.
4. Run a kernel to perform the necessary calculations, storing results in the allocated memory.
5. Copy the results back to a piece of host-memory for output or further processing.

In CUDA C, this looks like the following:

²This is a generalized statement. Scenario's exist for which it *is* more efficient to operate directly on (pinned) host-memory from the device. However, for algorithms that need to do multiple operations on a large portion of data, it is usually more efficient to copy this data to the device first and operate locally.


```

1  __global__ void kernel(int *buffer, int param1, int param2);
2
3  void host_code(int n) // take n to be the number of elements
4  {
5      // (1) allocate memory in RAM
6      int *host_mem = malloc(n * sizeof(int));
7
8      // (2) allocate memory in VRAM
9      int *device_mem;
10     cudaMalloc(&device_mem, n * sizeof(int));
11
12     // (3) initialize elements to 0
13     cudaMemcpy(device_mem, 0, n * sizeof(int));
14
15     // (4) call the kernel with 1xn threads
16     kernel<<< 1, n >>>(device_mem, 0, 1);
17
18     // (5) copy results back to host
19     cudaMemcpy(host_mem, device_mem, n * sizeof(int));
20
21     // ... further processing
22 }

```

Of course, the details of the implementation may vary wildly (using different means of allocating memory, etc), but the above snippet illustrates a much recurring structure in GPU programming.

5.4.3 Implementation of a Simple Kernel

Apart from the new syntax that features the triple pointy brackets (<<<...>>>), a kernel-call looks just like a regular function-call. The same holds for its implementation, which is like a function that is executed by many threads, perhaps concurrently. To illustrate what this may look like, consider this simple kernel that takes two vectors v_1 and v_2 of n elements, and calculates a third vector v_3 for which $v_3^i = v_1^i + v_2^i$:

```

1  __global__ void vectorAdd(double *v1, double *v2, double *v3)
2  {
3      // global thread-ID
4      int const tid = threadIdx.x + blockIdx.x * blockDim.x;
5
6      // Do the addition, using the global thread-ID as the index
7      v3[tid] = v1[tid] + v2[tid];
8  }

```

In the above kernel, the thread-ID acts as an index to the array. Because this index is unique to each thread (Figure 5.3), each will process a different element, maximizing the parallelism. However, this will only work when the number of threads dispatched is exactly equal to the number of elements. Neither is there any check to make sure that this resulting index falls within the range of the arrays. A common way to solve both these issues is to take the resulting thread-ID as a starting index, and then iterate over the array until the index is out of bounds.

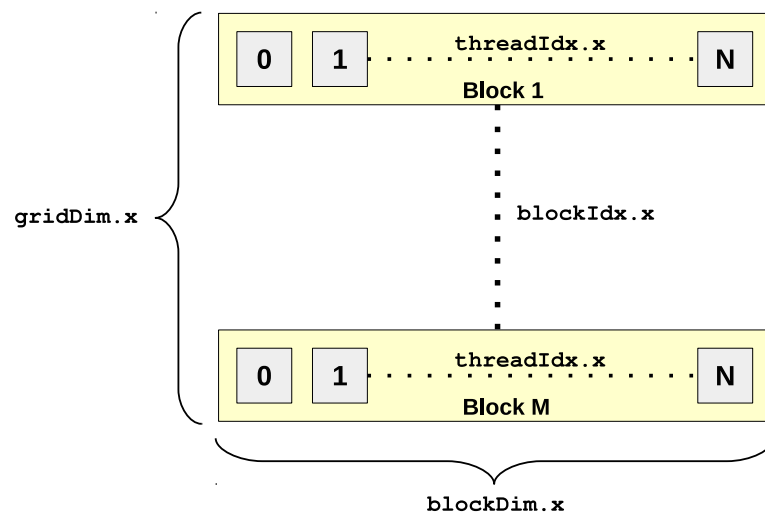


Figure 5.3: Schematic layout of threads within blocks in a grid. The global thread-ID k in a 1D grid (size M) of 1D blocks (size N) can be calculated as $k = i + jN$ where i and j denote the thread-index and block-index respectively.

```

1  __global__ void vectorAdd(double *v1, double *v2, double *v3, int n)
2  {
3      // Global thread-ID
4      int const tid = threadIdx.x + blockIdx.x * blockDim.x;
5
6      // Total number of active threads:
7      int const nThreads = blockDim.x * gridDim.x;
8
9      // Iterate over the array, starting at idx == tid
10     int idx = tid;
11     while (idx < n)
12     {
13         v3[idx] = v1[idx] + v2[idx];
14         idx += nThreads;
15     }
16 }

```

The above adaptations make sure that every pair of elements is handled, regardless of the number of threads dispatched. Even such a simple kernel requires some thought to make it failsafe. Moreover, it assumes that these vectors are already allocated and initialized in GPU memory, all of which requires work to have been done prior to calling the kernel. Then, when the work has been done, the results have to be copied back to the host for further processing, unless the result is only used by the GPU again.

5.4.4 Shared Memory and Barrier Synchronization

In more realistic and complex problems, data reduction is a common problem. For example, instead of adding two vectors together, assume we need to implement the dot-product of these vectors. The result is a single scalar value, so the data has to be reduced. This section will focus on the most common technique to achieve this, using shared memory.

The first step to calculating a dot-product is to multiply every corresponding element of the two vectors. This is very similar to the second example of Section 5.4.3, where the addition operation has been replaced by a multiplication. However, the third vector (`v3`) has become superfluous. Instead, each thread keeps a local value to store the sum of all its results:

```
1  double tmp = 0;
2  int idx = tid;
3  while (idx < n)
4  {
5      tmp += v1[idx] * v2[idx];
6      idx += nThreads;
7  }
```

When the code above has finished, each thread has a result stored in its local `tmp` variable. Now all that has to be done is to find the total sum of all these local values. This means that these values have to be communicated to each other, both at the block- and grid-level.

To calculate the block-local sum, we will use shared memory to share the local results among other threads in the block. To do so, a shared-memory block has to be declared (usually at the top of the kernel):

```
1  __shared__ double cache[THREADS_PER_BLOCK]; // static allocation
2  __shared__ extern double cache[]; // dynamic allocation
```

The code above shows two ways to do this: statically and dynamically. The static version requires that the number of elements in the cache is known at compile-time, for example by using a preprocessor `#define`, or making it a template parameter. The dynamic version does not specify a size at all. Instead, its size (measured in bytes) is passed to the kernel as a third launch parameter.

```
1  // static template-version
2  kernel<shared_size> <<<blocks, threads>>>(param1, param2, ...);
3  // dynamic version
```

```
4 kernel<<< blocks, threads, shared_size >>>(param1, param2, ...);
```

Once enough shared-memory has been allocated, each thread can store its `tmp` variable:

```
1 cache[threadIdx.x] = tmp;
2 __syncthreads();
```

After storing the local value in the shared array, the CUDA library function `__syncthreads()` is called, which provides a barrier-synchronization for all threads in the block, i.e. all threads that have access to the shared array. This is done to make sure that, once control passes through this point in the code, it has been guaranteed that each thread ‘sees’ an up-to-date version of the shared data.

The clever and more difficult part is up next. For starters, roughly half the threads will be disposed of. The half that remains active will then each add together two values of the shared data, reducing the data by a factor two. Again, the number of active threads will be divided in two to do the next addition and so forth. This will be repeated until only one single thread adds together the final pair of values, storing its result in the first element of the shared array. For simplicity, the number of threads per block will be assumed to be a power of 2:

```
1 int nActive = blockDim.x / 2;
2 while (nActive != 0)
3 {
4     if (threadIdx.x < nActive) // thread is still active
5         cache[threadIdx.x] += cache[threadIdx.x + nActive];
6     __syncthreads();
7     nActive /= 2; // half the number of active threads
8 }
```

Listing 5.1: A reduction algorithm in shared memory.

This will produce a partial result for each block. Consequently, each block should store this result somewhere in global memory for later reference, because values can not be reliably communicated between different blocks as there is no way for inter-block synchronization. Assuming a third parameter (e.g. `double *block_partial`), pointing to at least as many elements as there are blocks, one of the threads can copy the local result to this memory:

```
1 if (threadIdx.x == 0) // first thread
2     block_partial[blockIdx.x] = cache[0];
```

When the kernel is called and returns, the actual dot-product is still not available. As mentioned briefly before, CUDA does not provide a way of synchronizing between blocks. Consequently, data cannot be shared between blocks within the same kernel. Either the CPU should ask for the `block_partial` data and do the final summation itself, or another kernel (launched with just 1 block) is dedicated to performing the sum. When a second kernel is invoked to do the final reduction, it would be very similar to Listing 5.1. Had this task been left to the CPU, the host-code (including the kernel-call) would be along the lines of the following (using `std::vector` for convenience):

```

1 // assume v1, v2 and block_partial have been properly allocated
2 kernel<<< blocks, threads, threads * sizeof(double) >>>(v1, v2,
   block_partial);
3
4 vector<double> local_buffer(blocks);
5 cudaMemcpy(&global_buffer[0], block_partial, blocks * sizeof(double),
   cudaMemcpyDeviceToHost);
6
7 double result = 0;
8 for (double x: local_buffer)
9     result += x;

```

Again, we notice that even when making assumptions to simplify the kernel, a lot of overhead and error-prone coding is involved.

5.4.5 Caveats and Lessons Learned

While writing CUDA code, it is not uncommon for a program to compile and even run, but without giving proper, or any, output. This small chapter will be dedicated to a list of things to do and check before giving up and starting over.

1. When a kernel is launched with parameters incompatible with the hardware (too many threads, too much shared memory), there is nothing by default to notify the user that this has occurred. Therefore, during testing and debugging, it is good practice to invoke `cudaGetLastError`, pass the result to `cudaGetErrorString` and send the output to `stdout` or `stderr`. However, to make sure that the CPU waits for the kernel to return, call `cudaDeviceSynchronize` before requesting the error code:

```

1     kernel<<< ... >>> (...);
2     cudaDeviceSynchronize();
3     std::cerr << cudaGetErrorString(cudaGetLastError()) << '\n';

```

2. The above will at some point print out **unspecified launch failure**. This usually means that something went wrong while addressing memory in the kernel. In this case, check whether memory accesses are within bounds at all times.
3. If the kernel relies on shared memory, check whether threads have been synchronized appropriately. If the program fails to halt, make sure that the calls to `__syncthreads()` are placed outside conditional statements that may be unreachable to certain threads.
4. When communicating through shared memory, consider the use of *voting functions* (`__all()`, `__any()` and `__ballot()`) which allow for fast communication within a warp. Their use is limited to reducing predicates within a warp to a single integer, available to all threads

within this warp. In the case of `__all()` and `__any()`, this integer will evaluate to *true* (1) or *false* (0) when the predicate evaluates to true or false for all or any of the threads. The `__ballot()` function returns a 32-bit integer, of which each bit indicates whether the predicate evaluated to 0 or 1 for the corresponding thread. This could serve as an initial reduction, not requiring synchronization, prior to combining the results of different warps, but requires careful programming.

Chapter 6

Tracking & Hough Line Transformations

Even though this report is primarily concerned with vertexing rather than tracking, a particular technique known as the Hough transform will be evaluated in this chapter as an introduction to the power of GPU computing in practise. This technique is especially suitable for identifying lines in a set of points in 2D, but might be extendible to more dimensions.

6.1 Definitions

In general, the Hough transform can be used to recognize shapes in an image and to find the parameters that define them. This chapter will describe the method to reconstruct (straight) lines in an image like that of Figure 6.1. A line can be parameterized by two parameters. These parameters will be labeled (a, b) where a is the slope of the line and b is its offset, such that $y(x) = ax + b$. This defines a two-dimensional parameter-space in which every point represents a unique line. Thus, for each point in a (discretized) parameter-space, we can calculate which measurements could belong to the line defined by this point. However, when using (a, b) , the slope parameter a will tend to infinity as the line tends to become vertical. This would mean that a very large grid is necessary to represent a parameter space that covers all possible lines.

To avoid this issue, the alternative parameter-space (r, θ) is used to define a line. Here, r is the (shortest) distance from the origin to the line, and theta is the angle between the x -axis and r , such that

$$a = \frac{-1}{\tan \theta}, \quad b = \frac{r}{\sin \theta} \quad (6.1)$$

or alternatively

$$r = \frac{-b}{\sqrt{1 + a^2}}, \quad \theta = -\arctan\left(\frac{1}{a}\right). \quad (6.2)$$

Given (r, θ) , the distance from some point $P(x_p, y_p)$ to the line is then given by

$$d(r, \theta, \rho, \phi) = \rho \cos(\theta - \phi) - r \quad (6.3)$$

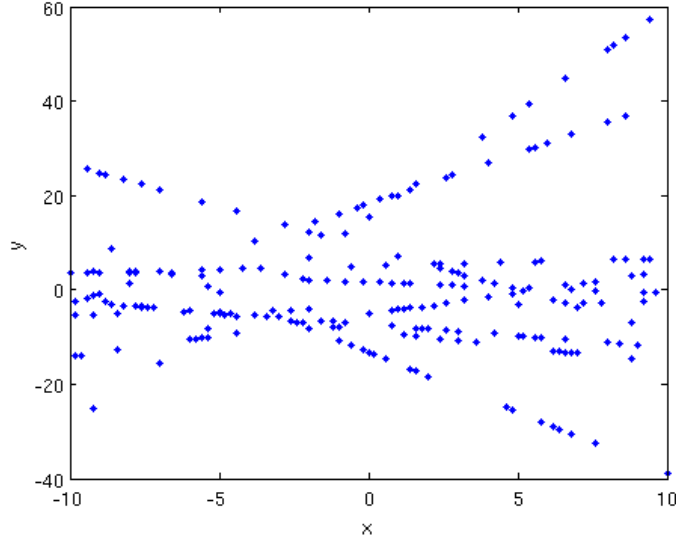


Figure 6.1: Example of a set of points taken from a set of lines that need to be reconstructed in 2D.

where $\rho = \sqrt{x_p^2 + y_p^2}$ and $\phi = \arctan\left(\frac{y_p}{x_p}\right)$ is the angle between P and the x -axis (Figure 6.2).

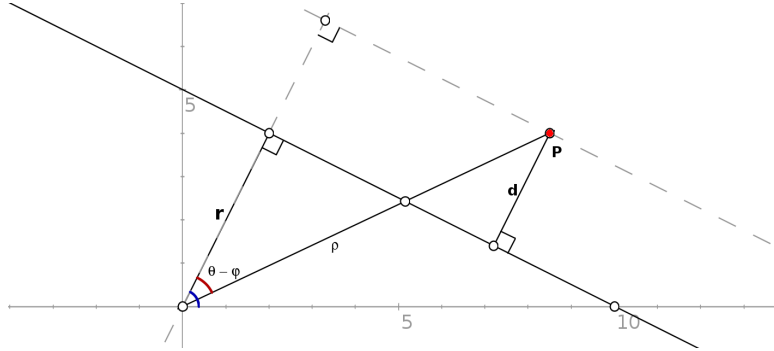


Figure 6.2: The line defined by $y = -x/2 + 5$ ($(a, b) = (-1/2, 5)$) can also be represented by its parameters (r, θ) , where $r = |\vec{r}|$ is the shortest distance between the origin and the line, and θ is the angle between \vec{r} and the x -axis.

A threshold ϵ is established such that any point P is said to be on the line when

$$d(r, \theta, \rho, \phi) < \epsilon. \quad (6.4)$$

Rather than a hard cut at ϵ , a continuous measure can be defined to quantify the probability with which a point can be said to be on the line.

$$f(r, \theta, \rho, \phi) = A \exp(-\alpha d^2), \quad (6.5)$$

where α is related to the grid-size and measurement-error, and $A = \left(\int \exp(-\alpha x^2) dx\right)^{-1}$ is the normalization constant.

6.2 Algorithm

A brute force HLT (Hough Line Transform) algorithm will perform a search for each measured point (x_p, y_p) across the entire grid to find the set of lines to which this point could belong. The rows of the grid (vertical direction) correspond to the angle θ and the columns (horizontal direction) correspond to the radius r , increasing with an amount $\Delta\theta, \Delta r$ respectively. When a match is found, the value of the cell corresponding to these parameters is incremented, yielding a 2D histogram. The cells that end up with the highest value correspond to the ‘true’ lines within our measurement.

```

1 // Given:
2 //     std::vector<Point> data: all datapoints (x, y)
3 //     double rMin:    minimum radius
4 //     double rMax:    maximum radius
5 //     int rRes:       resolution (#bins) in r
6 //     double thMin:   minimum angle theta
7 //     double thMax:   maximum angle theta
8 //     int thRes:      resolution (#bins) in theta
9 //
10 //     double distance(Point, double, double): see Eq 6.3
11 //     double epsilon(Point, double):          see Eq 6.8
12 //
13 //     A 2D indexable definition of HoughSpace.
14
15 HoughSpace result(rRes, thRes);
16 double dr = (rMax - rMin) / rRes;
17 double dth = (thMax - thMin) / thRes;
18
19 for (Point const &point: data)
20 {
21     double radius = rMin;
22     for (int col = 0; col != rRes; ++col)
23     {
24         double theta = thMin;
25         for (int row = 0; row != thRes; ++row)
26         {
27             if (distance(point, radius, th) < epsilon(point, th))
28                 ++result(row, col);
29             theta += dth;
30         }
31         radius += dr;
32     }
33 }
```

To define the parameter ϵ , the maximum distance-difference Δd_{max} between two adjacent cells

in Hough-space is considered. Two lines l_1, l_2 that have adjacent parameters in 2D Hough-space, have a distance d_1, d_2 with respect to some point (x_p, y_p) . The maximum difference between these two occurs when $l_1 = l_1(r, \theta)$ and $l_2 = l_2(r + \Delta r, \theta + \Delta \theta)$. The maximum difference can therefore expressed as

$$\Delta d_{max} = |d_2 - d_1| = |\rho [\cos(\theta - \phi + \Delta \theta) - \cos(\theta - \phi)] - \Delta r| \quad (6.6)$$

For small $\Delta \theta$ and using $f(x + h) - f(x) \approx h \frac{df}{dx}$, this can be rewritten to

$$\Delta d_{max} = \rho \Delta \theta |\sin(\theta - \phi)| + \Delta r. \quad (6.7)$$

This difference in distance from a point to two lines that are close in Hough-space defines the order of magnitude in which the parameter ϵ should be chosen. To ensure sufficient resolution, the value

$$\epsilon = \frac{\Delta d_{max}}{2}. \quad (6.8)$$

is chosen for the remainder of this chapter.

Applying this algorithm to the measurements of Figure 6.1 yields the Hough-space as displayed in Figure 6.3, where the brighter regions correspond to peaks in the grid. On closer inspection, 10 peaks can be distinguished by eye, which happens to be exactly the number of lines that were used to generate the data of Figure 6.1.

6.3 Peakfinding

Finding peaks in a 2-dimensional grid is a nontrivial task, and should be handled with care. A simple peakfinder can be defined that operates according to the following steps:

1. Given a 2D histogram H , a bin-threshold T and a (Euclidean) cell-distance threshold D ,
2. construct a list of all indices whose bin-value exceed T :

$$L = \{(i, j) \mid H(i, j) \geq T\}$$

```

1 // Given:
2 //   HoughSpace grid: the result of the hough-transform
3 //   int nRows:       number of rows in the grid
4 //   int nCols:       number of columns in the grid
5 //   int thresh:      cutoff parameter
6
7   std::vector<std::pair<int, int>> candidates;
8   for (size_t row = 0; row != nRows; ++row)
9       for (size_t col = 0; col != nCols; ++col)
10          if (grid(row, col) >= thresh)

```

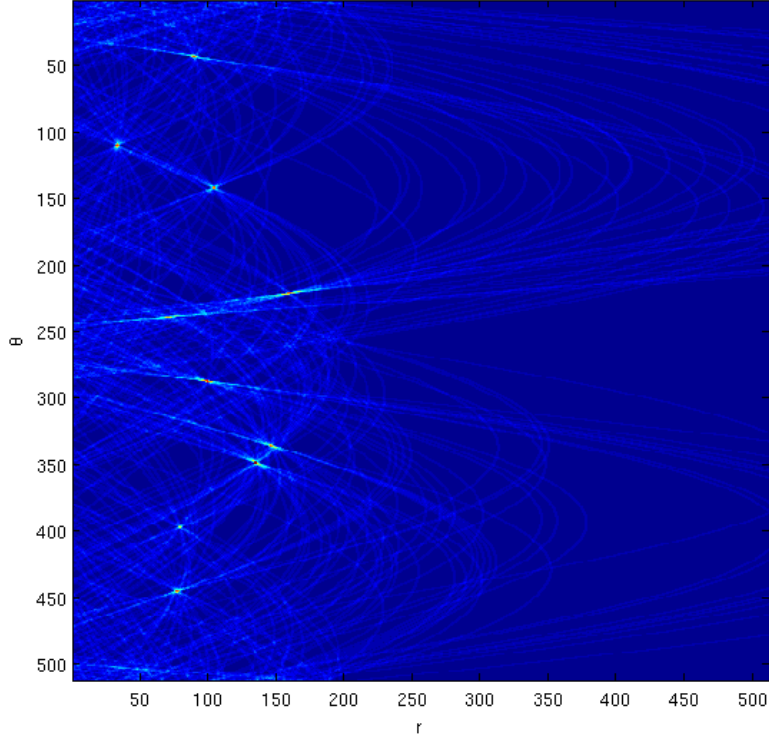


Figure 6.3: Hough Space

```

11         candidates.emplace_back(row, col);
12
13 // Note: the result in 'candidates' is denoted  $L$  above and below.

```

3. For every combination $\{(i, j), (k, l)\}$ of indices in L , find the Euclidean distance $d_{ij}^{kl} = \sqrt{(i - k)^2 + (j - l)^2}$, and compare this to D . If these cells are within D of each other, eliminate the one with the lowest histogram-count from L :

$$\forall (i, j, k, l) : \quad L \rightarrow L \setminus \underset{\{(i, j), (k, l)\}}{\operatorname{argmin}} H$$

```

1 // Given:
2 //     std::vector<std::pair<int, int>> candidates ( $L$  step 2)
3 //     double distThresh: Euclidean distance-threshold  $D$ 
4 //     double candidateThreshold(pair, pair): returns  $d_{ij}^{kl}$ 
5 //     void deleteCandidate(pair): marks candidate deleted in  $L$ 
6

```

```

7   for (int i = 0; i != candidates.size(); ++i)
8   {
9       for (int j = 0; j != i; ++j)
10      {
11          auto &x = candidates[i];
12          auto &y = candidates[j];
13          if (candidateDistance(x, y) < distThresh)
14          {
15              if (gridValue(x) <= gridValue(y))
16                  std::swap(x, y);
17
18              deleteCandidate(y);
19          }
20      }
21  }

```

4. Repeat from (3) until L contains no more cells within D of each other.
5. Define the elements left in L (not-deleted) as peaks in the histogram.

The algorithm above finds 10 peaks in the measurements presented. The lines to which these peaks correspond are drawn in Figure 6.4.

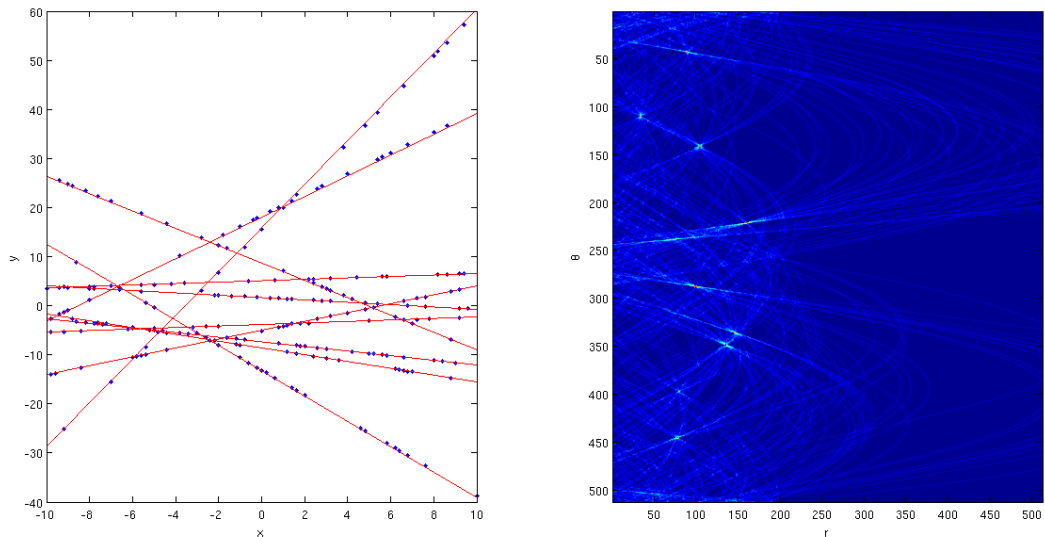


Figure 6.4: Result

6.4 Optimization

6.4.1 Paint by Numbers

Given the brute-force nature of the algorithm, it's not very surprising that the highly parallel GPU implementation outperforms the sequential CPU implementation. Instead of grabbing for a more powerful tool like a GPU, one should first try to sharpen the tools already at hand. When searching through Hough-space for lines that intersect some point $P_i = (x_i, y_i)$, the overwhelming majority of combinations (r, θ) will not yield a hit, making the algorithm highly inefficient. Instead of doing a brute-force search, it can be established analytically which parameters *do* correspond to lines that cross P_i . As can also be seen from e.g. Figure 6.3, each point P_i produces a curve in Hough-space that can be described by

$$r(\theta) = x_i \cos \theta + y_i \sin \theta. \quad (6.9)$$

Therefore, only one dimension of the parameter-space (e.g. θ , which is bound between 0 and 2π) had to be traversed. The curve can then be ‘painted’ directly into Hough-space, considerably simplifying the complexity of the algorithm. If N denotes the number of points P_i and R and Θ denote the number of bins in r and θ respectively, the complexity is reduced from $\mathcal{O}(NR\Theta)$ down to $\mathcal{O}(N\Theta)$, yielding a speedup of order $\mathcal{O}(R)$. Analogously, one could rewrite Equation 6.9 as $\theta(r)$ and achieve $\mathcal{O}(NR)$ complexity.

6.4.2 Analytical Approach

The above can be taken even one step further. Because the curves can be described analytically, so can the intersections of any pair of two curves. Given two points P_i and P_j , their curves intersect at

$$\theta_{\pm} = 2 \arctan \left(\frac{(y_i - y_j) \pm \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}}{x_i - x_j} \right), \quad (6.10)$$

where the sign is determined by the demand that $r(\theta) > 0$. Calculating the coordinates of this intersection in (r, θ) -space is effectively the same as constructing a straight line through the pair of datapoints. The (r, θ) parameters of this line (Equation 6.2) are the coordinates of the intersection.

Realizing the above, one is able to find the intersection-coordinate of two curves in Hough-space a more straightforward way. So far, only the length of the r vector has been considered. In vector representation, it can be written as

$$\vec{r} = \frac{b}{1 + a^2} \begin{pmatrix} -a \\ 1 \end{pmatrix}. \quad (6.11)$$

The angle $\theta \in [0, 2\pi]$ of this vector is calculated in the snippet below using `std::atan2`¹. This function will determine the sign of the angle based on the quadrant, which is derived from its

¹The `atan2`-function is found in most programming-environments and/or libraries.

arguments. For every combination of datapoints, the (r, θ) -parameters corresponding to the line intersecting both points can be calculated. A bin in the 2D Houghspace histogram is then incremented in order for the peakfinding algorithm to identify the bins with the most hits.

```

1 // Given:
2 //     std::vector<Point> data: all datapoints (x, y)
3 //     double rMin:    minimum radius
4 //     double rMax:    maximum radius
5 //     int rRes:       resolution (#bins) in r
6 //     double thMin:   minimum angle theta
7 //     double thMax:   maximum angle theta
8 //     int thRes:      resolution (#bins) in theta
9 //
10 //     A 2D indexable definition of HoughSpace.
11
12 HoughSpace result(rRes, thRes);
13 double dr = (rMax - rMin) / rRes;
14 double dth = (thMax - thMin) / thRes;
15
16 for (size_t i = 0; i != data.size(); ++i)
17 {
18     for (size_t j = 0; j != i; ++j)
19     {
20         // Calculate (r, theta)-coordinates
21         double dx = data[i].x - data[j].x;
22         double dy = data[i].y - data[j].y;
23         if (dx == 0.0)
24             continue; // ignore
25
26         double a = dy / dx;
27         double b = data[i].y - a * data[i].x;
28         double q = b / (1 + a * a);
29
30         double r = std::sqrt(b * q);
31         double th = std::atan2(q, -slope * q);
32         if (th < 0)
33             th += 2 * M_PI;
34
35         // Convert to (row, col)-pair
36         int row = (th - thMin) / dth;
37         int col = (r - rMin) / dr;
38         int val = ++result(row, col);
39     }
40 }

```

The downside of this approach is that now the complexity has become a stronger function of the data-size N rather than the gridsize $R\Theta$. A set of N datapoints produces a total of $N(N-1)/2$ pairs, for each of which the point of intersection has to be calculated. The algorithmic complexity therefore becomes $\mathcal{O}(N^2)$.

6.5 Generalization to multi-D

Thus far, the Hough transformation has been applied to lines in the 2-dimensional (x, y) - or (r, θ) -plane. The hits (x_i, y_i, z_i) inside the VELO live in 3D space such that the tracks can be constrained by a total of 4 parameters (a, b, c, d) :

$$\begin{aligned}x &= a + bz \\ y &= c + dz.\end{aligned}$$

Therefore, rather than scanning through a 2D grid of $\mathcal{O}(100^2)$ cells (taking R and Θ to be of $\mathcal{O}(100)$), the brute-force search algorithm has to scan a 4D grid of $\mathcal{O}(100^4)$ cells. Even with the magnificent power of modern GPU's, this approach does not seem reasonable.

The semi-analytical approach scales much better to higher dimensions. As with lines in 2D, it is a fairly trivial exercise to construct the parameters of a line through 2 points in 3D. The computational complexity of constructing the Houghspace histogram (still $\mathcal{O}(N^2)$) is independent of its dimensions. The result is however still a 4-dimensional histogram which has to be searched, but this is a problem that exists for both the brute force and analytical method and might even be overcome by hashing these parameters down into a lower dimensional problem, such that only a 1D-histogram has to be handled.

6.6 Performance Comparison

To illustrate the differences in both scaling and absolute performance, the naive and analytical algorithms have been implemented on both the CPU and GPU. In the first benchmark, the size of the (square) Hough-space was varied between 10^2 and 512^2 bins in order to compare the dependence on R and Θ for both algorithms. The second benchmark was run on a constant 128×128 Hough-space, varying the number of available datapoints. The sample contained 20 (hidden) tracks that were reconstructed from sets varying in size between 10 and 10000 datapoints. The benchmarks below were run on an Intel i5-6600 (3.9GHz) CPU and the NVidia GTX970 GPU.

6.6.1 Grid Dependence

Figure 6.5 shows the performance of the abovementioned implementations for varying gridsizes. As predicted, the complexity scales with M^2 (where $M = R = \Theta$ is the gridsize). For large enough grids, the GPU becomes saturated when running the naive algorithm. When this

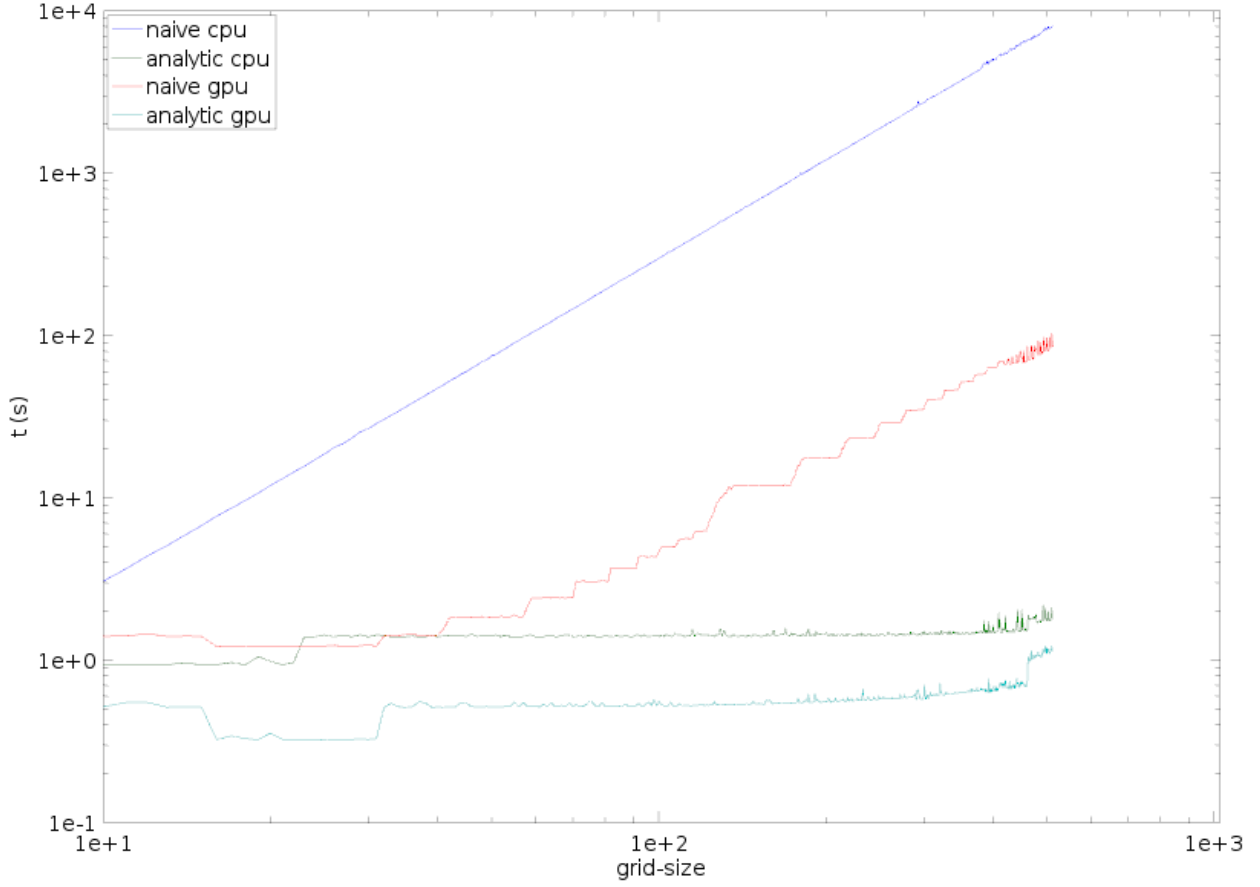


Figure 6.5: Benchmark results of 4 competing implementations to generate the Hough-space histogram. In this benchmark, the (square) grid-size was varied between 10^2 and 512^2 elements.

happens, the scaling properties of the GPU performance become similar to that of the CPU running a single thread.

The analytic method scales only with N^2 (where N is the number of hits), which is visible in Figure 6.5 as (nearly) constant runtimes for both implementations. Even here, the GPU outperforms the CPU because it has been parallelized over every independent hit-combination (for a total of $N(N - 1)$ combinations). In this benchmark, N was kept constant at $N = 200$.

6.6.2 Hit Dependence

In the second set of benchmarks, the gridsize has been kept constant at 128×128 bins. Instead, the number of hits was varied between 10 and 10000 (Figure 6.6).

Indeed, the naive brute-force methods scale as expected with N , where the GPU version shows a slightly smaller slope when fully saturated due to its many parallel computations.

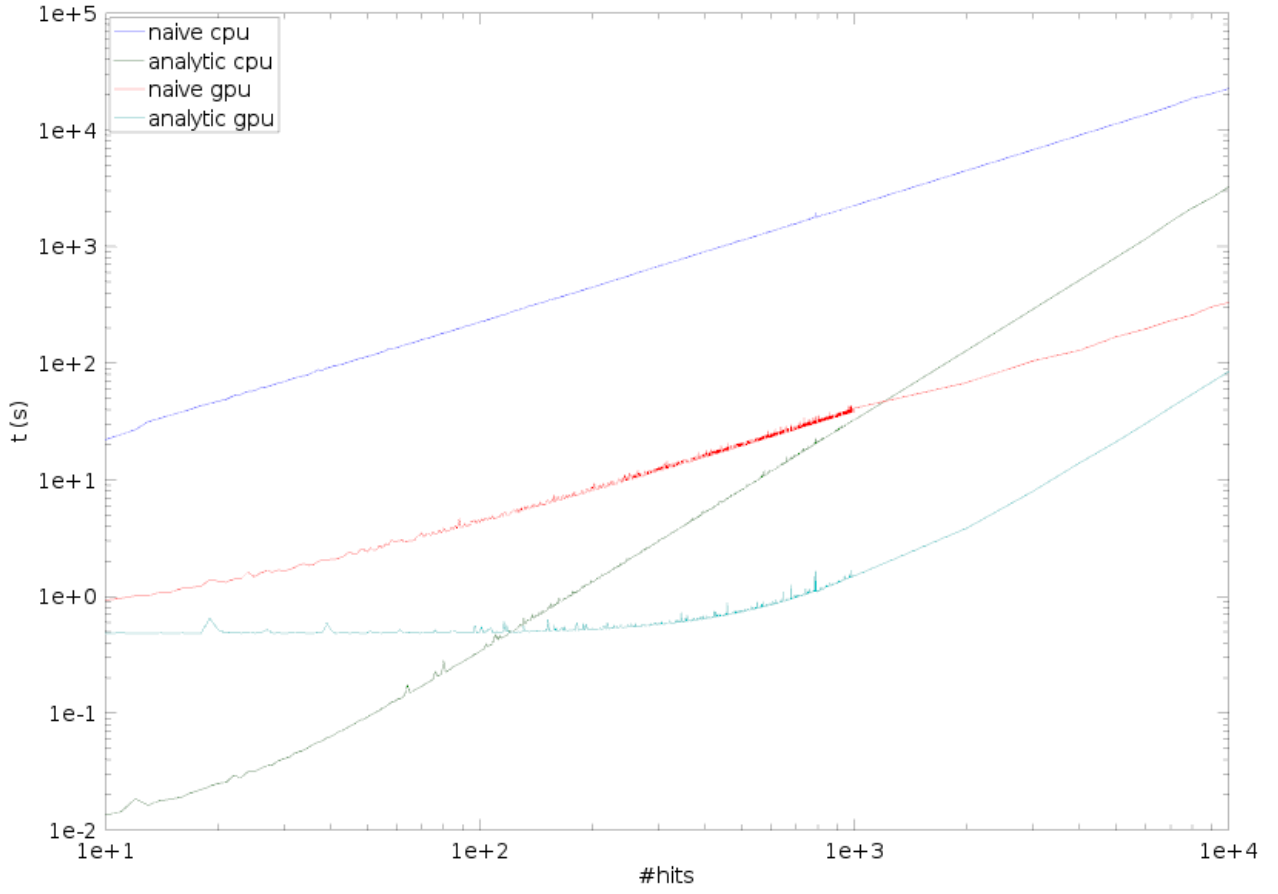


Figure 6.6: Benchmark results of 4 competing implementations to generate the Hough-space histogram. In this benchmark, the number of datapoints used to generate the histogram was varied between 10 and 10,000 elements.

Interestingly, the CPU is preferred for smaller datasets when using the analytical approach. This initial advantage quickly vanishes when the dataset exceeds ~ 100 points due to the N^2 complexity of this algorithm.

Chapter 7

Adjacency Graphs for Vertex Finding

7.1 Introduction

An adjacency matrix A is a matrix in which every matrix-element a_{ij} quantifies the relationship between two entities i and j . These entities are often called *nodes* or *vertices*, but given the ambiguous nature of the latter term in the context of this work, the former is adopted. In binary adjacency matrices, the only information contained in the matrix is whether or not two elements are related (adjacent): $a_{ij} \in \{0, 1\}$. In weighted matrices, the elements can have values in some pre-specified range, e.g. $a_{ij} \in [0, 1]$. Furthermore, adjacency matrices can be either directed or undirected. Undirected matrices are symmetric, such that $a_{ij} = a_{ji}$ whereas this property need not hold in directed matrices.

Adjacency matrices can be visualized using (adjacency-)graphs. The nodes are labeled and drawn as dots, and are being connected through lines when they are considered adjacent according to the adjacency matrix. Oftentimes, the position of the nodes in 2D-space is determined by regarding every connection as a spring-like force and letting the system converge to a state of equilibrium. This allows for clusters to come together, making it easier to identify these features by eye.

For example, the graph-representation of the following matrix is shown in Figure 7.1:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad (7.1)$$

7.2 Converting an Event to a Matrix

When buckets collide in the VELO, a shower is produced that generates hits in the different layers (Section 3). The tracking algorithm (Section 6) then combines these hits into tracks,

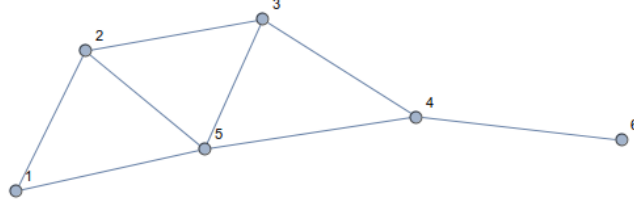


Figure 7.1: Graph of the example matrix (7.1).

or lines in 3D space. For the current analysis, only 4 parameters are of importance, being $\{x, y, t_x, t_y\}$. The (x, y) coordinate is the intersection of the line with a plane at some z -coordinate. These parameters therefore are functions of z . The t_x and t_y represent the slopes in the x and y directions:

$$t_x = \frac{p_x}{p}, \quad t_y = \frac{p_y}{p} \quad (7.2)$$

Assuming straight lines (ignoring scattering events for the moment), the t_x and t_y are constant in z , such that x and y become functions of z :

$$x(z) = x_0 + z t_x \quad (7.3)$$

$$y(z) = y_0 + z t_y. \quad (7.4)$$

The goal of interest given a set of tracks $T_i = \{x, y, t_x, t_y\}_i$ (where $i = 1, \dots, N_{\text{tracks}}$), is to find the common origin of certain subsets of tracks. It is unknown in advance which subsets originate from a common origin, or where these origins are located. This is where the concept of *adjacency* is introduced, and defined as follows: a pair of tracks (T_i, T_j) is considered *adjacent* if, at their point of *closest approach*, they are separated by an amount less than the threshold Δ from each other. The adjacency-matrix A contains boolean values that denote whether this holds true for every possible combination in the set:

$$A = \{a_{ij}\}, \quad \text{where} \quad (7.5)$$

$$a_{ij} = \begin{cases} 1, & \text{if } d_{ij} < \Delta \\ 0, & \text{otherwise} \end{cases} \quad (7.6)$$

The result is a symmetrical $n \times n$ adjacency-matrix that forms the basis for further analysis. For the calculation of d_{ij} see Section 7.4

7.3 Visualizing Events using Graphs

Visualizing entire events in terms of graphs is a worthwhile exercise because it allows for intuitive analysis of the complexity of the event, and whether or not operations on the graph have had their effect. For example, the event in Figure 7.2 was generated randomly by generating 6

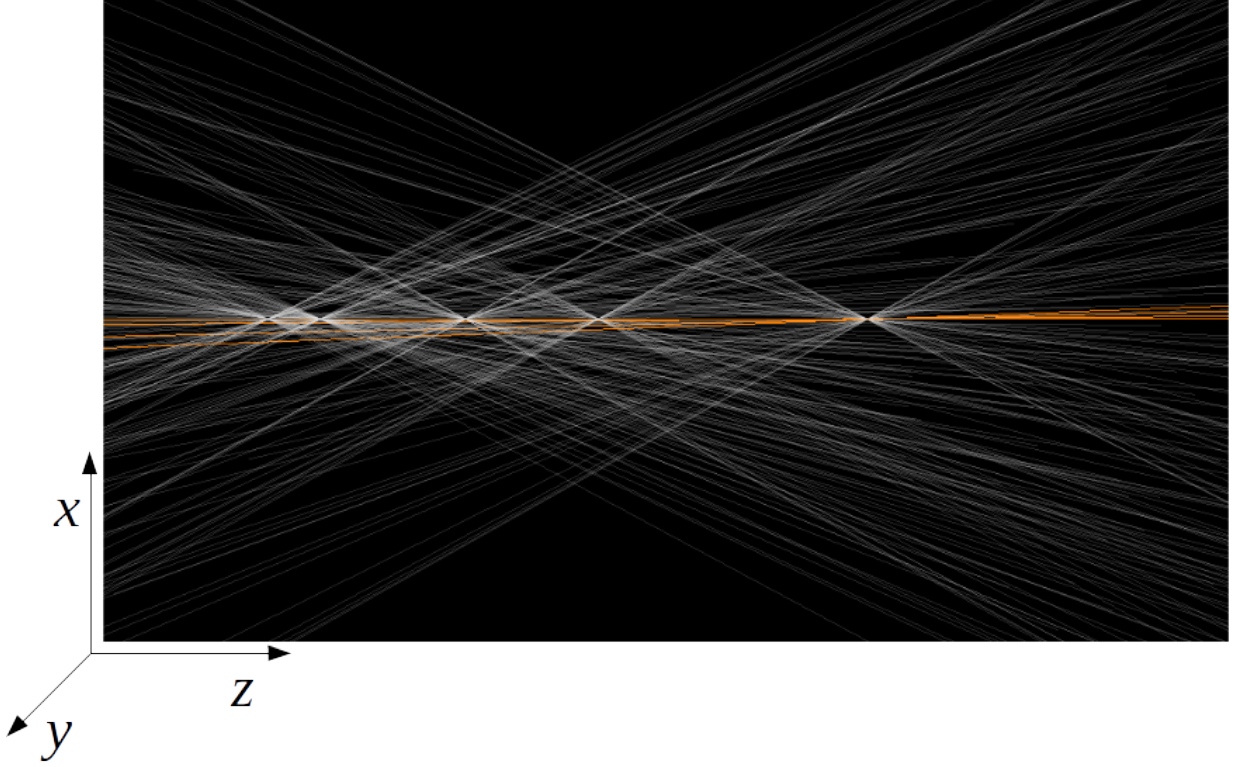


Figure 7.2: Randomly generated example event with 6 PV's and one SV (highlighted in orange).

primary vertices with a random number of tracks (according to a Poisson-distribution with $\mu = 50$) and a secondary vertex from which 3 secondary tracks is originating (highlighted in orange). Note that in the visualization of this event (Figure 7.2) tracks are pointing in both ways, that is, directional information is absent in the current representation of tracks. The graph corresponding to this event is shown in Figure 7.3, from which one can clearly make out at least 5 clusters, corresponding to the primary vertices. However, on closer inspection, one of the clusters seems to be a merger of two primary vertices that are so close together that many of the tracks that originate from different vertices still meet within Δ . Also, somewhere within this graph there should be a triplet of nodes that maps to the displaced vertex. The challenge is to weed this information out, or at least eliminate as many other tracks as possible.

7.4 Construction

To construct the adjacency-matrix of an event, the minimum distance between every pair of tracks needs to be considered. To calculate d_{ij} , the track is modeled as a line in 3D parameterized by two vectors (\mathbf{a}, \mathbf{b}) , where $\mathbf{a} = (a_x, a_y, a_z)$ describes its slope in all directions and

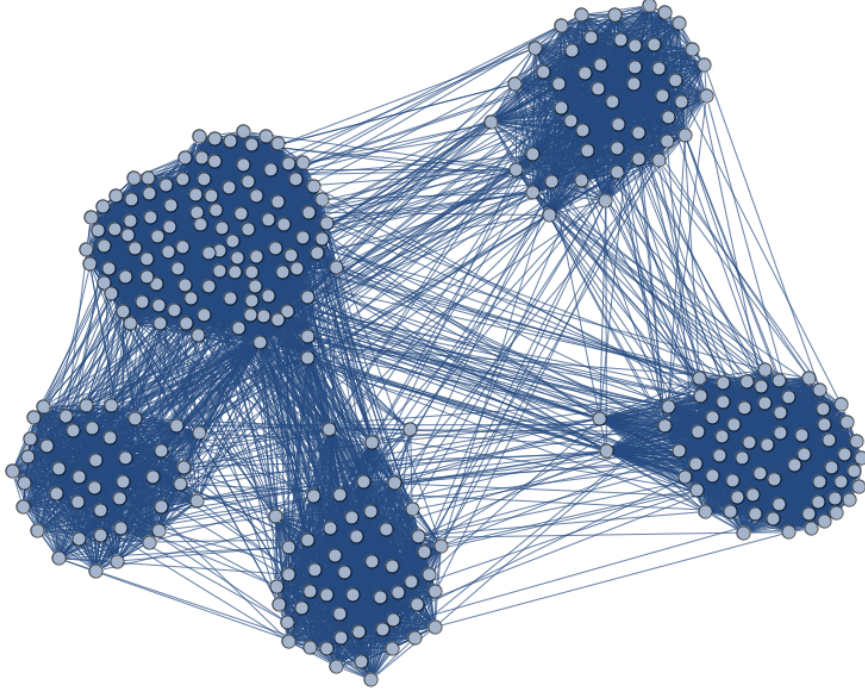


Figure 7.3: Adjacency graph from the event shown in Figure 7.2.

$\mathbf{b} = (b_x, b_y, b_z)$ is its offset (coordinates at $t = 0$).

$$\mathbf{r}(t) = \mathbf{a}t + \mathbf{b} \quad (7.7)$$

Given two lines \mathbf{r}_i and \mathbf{r}_j , the shortest vector that connects these lines is perpendicular to both, hence the unit vector $\hat{\mathbf{d}}_{ij}$ can be found by taking the outer-product of the slope-vectors and normalizing:

$$\hat{\mathbf{d}}_{ij} = \frac{\mathbf{a}_i \times \mathbf{a}_j}{\|\mathbf{a}_i \times \mathbf{a}_j\|} \quad (7.8)$$

The projection of an arbitrary vector between any two points on \mathbf{r}_i and \mathbf{r}_j onto $\hat{\mathbf{d}}_{ij}$ is therefore of the required length d_{ij} . For any t , this results in

$$d_{ij} = \left| \frac{\mathbf{a}_i \times \mathbf{a}_j}{\|\mathbf{a}_i \times \mathbf{a}_j\|} \cdot (\mathbf{b}_i - \mathbf{b}_j)^T \right| \quad (7.9)$$

Given that the calculation of d_{ij} for all (i, j) -pairs is an operation of $\mathcal{O}(n^2)$ and that each calculation can be performed independently, the GPU seems an excellent computing device to carry out these computations.

7.4.1 Accounting for Statistical Error

The track reconstruction does not only yield track parameters, but also the errors on these parameters. Rather than straight lines, tracks can thus be viewed as cones, where the radius of this cone at each point is a measure of the error. This can be incorporated into the adjacency-matrix by evaluating the maximum probability of coming within a certain distance of one another. This will result in different adjacency-graphs, but this additional complexity was not implemented in the course of this research.

Chapter 8

Adjacency Matrix Implementation and Performance

In the (C++) implementation of the adjacency-matrix construction, the notion of a track (or line in 3D) is represented by a simple POD (Plain Old Data) structure that can be used for both the CPU and GPU implementation and reflects the parameterization of 3D lines as used in Section 7.4:

```
1 struct Track
2 {
3     double a[3];
4     double b[3];
5 };
```

We can now define the distance function that implements Equation 7.9 and can be used both by the CPU and GPU implementation (as a `__device__` function, i.e. a subfunction that can be called only from GPU-kernels).

```
1 double distance(Track const &t1, Track const &t2)
2 {
3     // (b1 - b2)
4     double x[3] =
5     {
6         t1.b[0] - t2.b[0],
7         t1.b[1] - t2.b[1],
8         t1.b[2] - t2.b[2]
9     };
10
11     // (a1 x a2)
12     double y[3] =
13     {
14         t1.a[1] * t2.a[2] - t1.a[2] * t2.a[1],
15         t1.a[2] * t2.a[0] - t1.a[0] * t2.a[2],
```

```

16         t1.a[0] * t2.a[1] - t1.a[1] * t2.a[0]
17     };
18
19     // (b1 - b2) . (a1 x a2)
20     double dotxy = x[0] * y[0] + x[1] * y[1] + x[2] * y[2];
21     double normy = sqrt(y[0] * y[0] + y[1] * y[1] + y[2] * y[2]);
22
23     // d_{12}
24     return abs(dotxy / normy);
25 }

```

8.1 CPU

When using dynamically sized arrays of any kind in a standard C++ environment, a choice has to be made with respect to memory management. The C++ standard-library provides optimized facilities like `std::vector` to handle memory management in its entirety, including run-time allocation and dynamic resizing. Given that this problem does not require specialized functionality of the underlying datastructure, the `std::vector` can be used as-is. However, for generality, a type `AdjacencyMatrix` will be assumed to exist in the code that follows. In practise, this type may be defined as simple as

```

1 typedef std::vector<std::vector<bool>> AdjacencyMatrix;

```

When a simple `typedef` as above is used, there are no checks on the runtime integrity of its data. For example, when the implementation uses the above definition, it is possible to have $A(i, j) \neq A(j, i)$, even though the matrix is symmetric by definition. Similarly, as the matrix is hollow ($A(i, i) = 0$), diagonal elements should never be allowed to be assigned a value. It's a matter of implementation detail as to whether these constraints are implemented in the final product.

Given the above definitions and a list of tracks, the unoptimized construction of the matrix is fairly straightforward:

```

1 AdjacencyMatrix constructAdjacency(std::vector<Track> const &tracks,
2                                   double threshold)
3 {
4     size_t const N = tracks.size();
5
6     AdjacencyMatrix result = makeAdjacencyMatrix(N);
7     for (int i = 0; i != N; ++i)
8         for (int j = 0; j != N; ++j)
9             result[i][j] = (i != j)
10                ? distance(tracks[i], tracks[j]) < threshold
11                : 0;

```



```

12 |
13 |     return result;
14 | }

```

It should be obvious that this can be optimized by iterating not over every element of the matrix, but only its lower (or upper) triangle. This reduces the total number of operations from N^2 to $N(N - 1)/2$, producing a speedup of a factor ~ 2 with respect to the naive implementation. However, for illustrative purposes, the naive version is kept as a mold for the implementation on a GPU which is discussed in the following section.

8.2 GPU

Every operation in the innermost loop of the listing in the previous section is completely independent from every other operation. Thus, these calculations can be parallelized on the GPU. The goal is therefore to construct a kernel that fills the entire matrix with the appropriate values, regardless of its launch-parameters. E.g. it should not matter whether the kernel is launched with 1, 2, N or N^2 threads, or how these threads are distributed among the blocks. Its implementation then becomes similar to that of Section 5.4.3:

```

1  __global__ void constructAdjacency(bool *matrix, Track const *tracks,
2                                     int n, double thresh)
3  {
4      int nThreads = blockDim.x * gridDim.x;
5      int tid = threadIdx.x + blockIdx.x * blockDim.x;
6
7      int N = n * n;
8      int idx = tid;
9
10     while (idx < N)
11     {
12         int row = idx / n;
13         int col = idx % n;
14
15         matrix[idx] = (row != col)
16             ? (distance(tracks[row], tracks[col]) < thresh)
17             : 0;
18
19         idx += nThreads;
20     }
21 }

```

Every thread calculates its own thread-id from its local index, and the index of the block that it lives in. This thread-id then acts as a start index to a linear array in which the matrix elements are being stored. From this linear index, the corresponding row and column are calculated,

which map to the pair of tracks between which the distance has to be calculated and compared to the threshold. The result is then assigned to the linear array, and the thread increments its index by the total number of threads to handle the next element, as long as this still falls within the bounds of the array. Diagonal elements are skipped.

8.2.1 Memory

Whereas in the CPU implementation, memory management was as easy as using `std::vector`, there is no standard way of doing this on the GPU. The Nvidia toolkit however ships with a library called Thrust, which provides STL-like containers and algorithms for easy access to GPU performance. For example, `thrust::device_vector` can be used to manage memory on the device much like a `std::vector` manages memory on the host (in addition to which Thrust provides `thrust::host_vector`). Using such facilities alleviates the need to use functions like `cudaMalloc()`, which often lead to code bloat and introduce pitfalls to the programmer. Whichever method is used, the kernel above assumes that `matrix` points to an array of `n * n` elements of type `bool`. Even when using Thrust, raw pointers can be easily extracted from a `device_vector` and passed to kernels expecting such parameters.

8.3 Optimization

8.3.1 CPU

Ideally, every pair of tracks is considered exactly once. The naive implementations presented above will calculate every matrix element individually even though the matrix is symmetric and hollow. In the CPU implementation this is easily resolved by iterating over the lower-triangle only:

```

1 // assume the adjacency matrix has been initialized to 0
2 for (int i = 0; i != N; ++i)
3     for (int j = 0; j != i; ++j) // stop at i
4         result[i][j] = result[j][i] = // assign to both elements
5             distance(tracks[i], tracks[j]) < threshold;

```

Note that this approach saves cycles only, not memory, by calling the distance function $N(N - 1)/2$ instead of N^2 times and still storing the full symmetric matrix in memory.

8.3.2 GPU

In the context of a GPU, the transformation is less transparent. Whereas in the naive version it was completely clear which element should be accessed by each thread, a more complex mapping is required in the lower-triangular case (Figure 8.1). This figure shows the mapping thread-id's to elements of a 5×5 symmetrical matrix.

Because every i^{th} row is preceded by

$$\sum_{i'=0}^{i-1} i' = \frac{i(i-1)}{2}$$

elements, it must hold that the thread-ID k is related to the row i and column j as

$$k = \frac{i(i-1)}{2} + j. \quad (8.1)$$

Therefore, solving for i and j under the condition that $\frac{i(i-1)}{2} \leq k < \frac{i(i+1)}{2}$, the mapping from thread-ID to a row/column-pair becomes

$$i = \left\lfloor \frac{\sqrt{1+8k+1} + 1}{2} \right\rfloor \quad (8.2)$$

$$j = k - \frac{i(i-1)}{2}, \quad (8.3)$$

where $\lfloor \cdot \rfloor$ denotes the floor-function. This mapping has to be used by every thread to determine which pair of tracks to act on. Other than this, not much changes in the GPU implementation:

```

1  __global__ void constructAdjacency(bool *matrix, Track const *tracks,
2                                     int n, double threshold)
3  {
4      int nThreads = blockDim.x * gridDim.x;
5      int tid = threadIdx.x + blockIdx.x * blockDim.x;
6
7      // The total number of elements has changed
8      int N = n * (n - 1) / 2;
9      int idx = tid;
10
11     while (idx < N)
12     {
13         // Lower-triangular mapping
14         int row = (sqrt(1.0 + 8 * idx) + 1) / 2;
15         int col = idx - (row * (row - 1)) / 2;
16
17         // A single call to distance() to populate 2 matrix-elements
18         // (no need for diagonal checks)
19         matrix[row * n + col] = matrix[col * n + row] =
20             distance(tracks[row], tracks[col]) < threshold;
21
22         idx += nThreads;
23     }
24 }
```

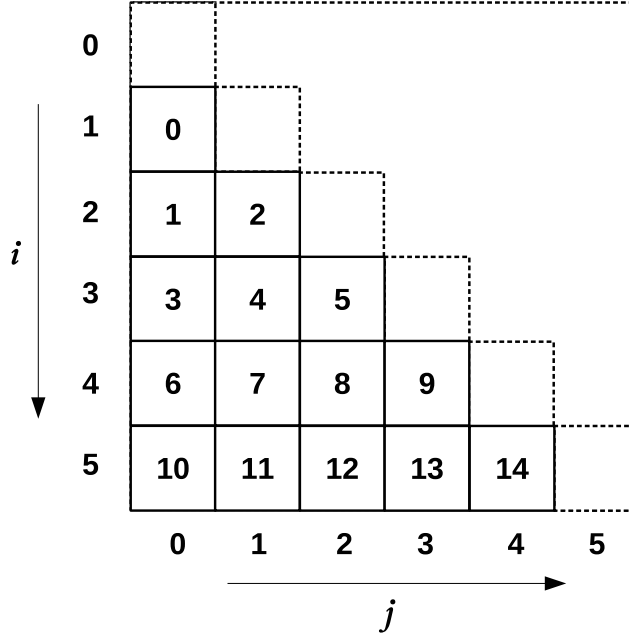


Figure 8.1: Mapping of thread-id's (numbers in the squares) to lower-diagonal elements of a (symmetric) matrix to reduce the memory footprint of a kernel operating on such matrices.

8.3.3 Memory

A further optimization could be done with respect to memory usage, as it might be wasteful to have every upper triangular element stored in addition to its lower triangular counterpart. Moreover, rather than two accesses per thread per iteration to global memory (which in most cases probably do not both reside in cache due to them being far apart in the linear representation of the matrix), the algorithm only needs one access. In a kernel that is memory-access intensive, such considerations could make a big difference.

The downside of this additional memory optimization is that future algorithms acting on the matrix will have to deal with the mapping involved, making these algorithms less transparent. Of course, this can all be abstracted away to still provide a workable interface to the programmer, but the mapping has to be computed prior to every memory access. Whether or not this is acceptable, depends on the algorithm at hand.

8.4 Benchmarking

All versions discussed above have been implemented and benchmarked. Figures 8.2 and 8.3 show the runtime-results, averaged over 100 runs, of the different versions of the `constructAdjacency` kernel for events of size $N \in [10, 418]$ and $N \in [100, 4100]$ respectively. The CPU implementation scales exactly with N^2 across the entire range of matrix sizes N , but the same cannot be said of the GPU implementations. This is especially obvious in Figure 8.3, where for very

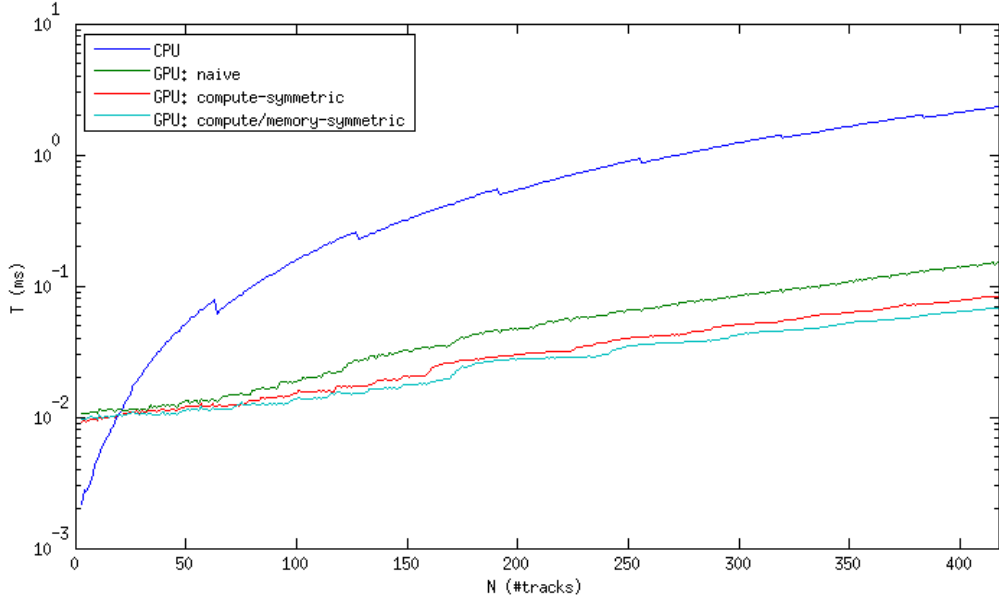


Figure 8.2: Performance of different algorithms to compute the adjacency-matrix on both the CPU and GPU for a relatively low (1-400) amount of tracks.

small matrices the curve is rather flattened before following the same N^2 dependency. This shows that the GPU can, up to some point, process the matrix in parallel. After this point, the device is fully saturated and runtime scales with N^2 just as one would expect from any other compute device.

As expected, the optimized versions outperform the naive version by a significant amount. Especially in the region where performance scales with N^2 , the ratio between the naive and optimized kernels becomes large, approaching

$$\lim_{N \rightarrow \infty} \frac{N^2}{N(N-1)/2} = 2. \quad (8.4)$$

Even the resources of a massively parallel device like the GPU are saturated by that point, and instructions have to be serialized anyway. However, for smaller matrices, even with $N \sim 100$, the GPU timings scale different from N^2 , which is where most of the speedup is gained. Luckily, events in the VELO consist of several hundreds of tracks at most, making the calculation of the adjacency-matrix very suitable for the GPU.

The difference between the two optimized kernels (compute-symmetric vs compute/memory-symmetric) can be solely attributed to memory effects. Not only does the latter version need only half the global memory-fetches, but the number of cache-misses should also be lower to the change in indexing. This effect is shown in Figure 8.4 and 8.5, where the different colors represent accesses by different warps. In this example, a 20x20 matrix is accessed by 6 warps ($6 \times 32 = 192$ threads), where every thread needs access to 2 matrix elements (i, j) and (j, i) . In the naive implementation, memory accesses are very delocalized and little to no performance

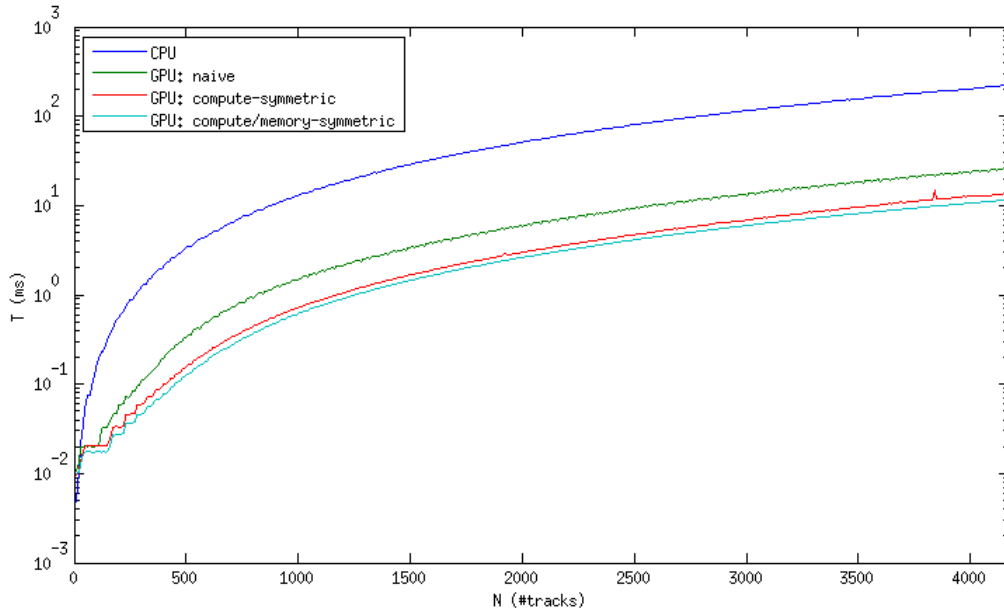


Figure 8.3: Performance of different algorithms to compute the adjacency-matrix on both the CPU and GPU for large (1-4000) amounts of tracks.

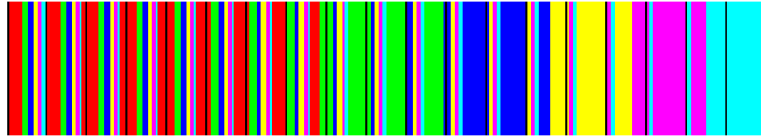


Figure 8.4: Distribution of matrix elements in memory when the full matrix is stored in memory, as used in the naive and flop-optimized version of the adjacency construction kernel.

can be gained from caching. In the optimized version (Figure 8.5), not only has the total memory usage been dropped from N^2 to $N(N-1)/2$ elements, but the elements are also better aligned in memory with respect to the warps that need access to them. This results in fewer cache misses and therefore faster access.

8.4.1 Caveat

When these benchmarks were performed on such large matrices, it came to light that one needs to be careful when allocating this many resources. A collection of $4000^2 = 16,000,000$ integers (64 MB of VRAM) can be easily stored in the global memory of a device capable of storing several gigabytes, but there are other factors that come in to play, especially when the number of blocks and threads is calculated at runtime. For example, the number of blocks per dimension in a grid is limited to $2^{16} - 1 = 65535$, and the number of threads per block is limited to 1024. When kernels are launched with parameters that exceed these limitations, the kernel is not launched at all, without reporting an error. The error that occurs can be retrieved, but this has to be done

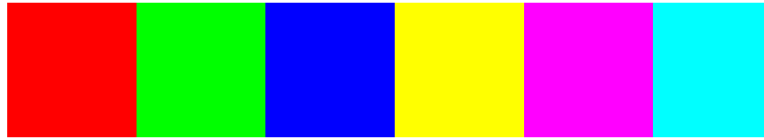


Figure 8.5: Distribution of matrix elements in memory when using the memory-optimized kernel.

explicitly using `cudaGetLastError()`, `cudaPeekAtLastError()`, `cudaGetErrorString()`, and combinations of these. A canonical way is to wrap each API-call that returns a `cudaError_t` in a macro like the example below:

```

1 #define CUDA_CHECK_ERROR(call) \
2     myCudaErrorCheck((call), __FILE__, __LINE__)
3
4 inline void myCudaErrorCheck(cudaError_t err, char const *fname,
5                             int line)
6 {
7     if (err != cudaSuccess)
8         cerr << "Error in " << fname << ", line " << line
9             << ":\n\t" << cudaGetErrorString(err) << '\n';
10 }

```

Now, every call like

```

1 CHECK_CUDA_ERROR( cudaMalloc(&buf, n * sizeof(int)) );

```

is wrapped in a safety-net. In the case of kernels (which do not return anything to the host), it is necessary to call `cudaGetLastError()` or `cudaPeekAtLastError()` to check whether everything went right:

```

1 kernel <<< b, t >>>();
2 CUDA_CHECK_ERROR( cudaGetLastError() ); // launch error
3 CUDA_CHECK_ERROR( cudaDeviceSynchronize() ); // execution error

```

To avoid issues caused by the limited device resources, one can use the `cudaGetDeviceProperties()` function to request a struct of the type `cudaDeviceProp` that holds properties like the maximum number of threads that can be launched per block, the maximum amount of shared memory, etc. Implementing these checks will catch most errors before the kernel is even launched and make the program portable to different CUDA hardware.

Chapter 9

Histogramming in Parallel

9.1 Introduction

The currently used method for finding primary vertices has been discussed in Section 3.3. Because the nature of this algorithm is inherently sequential, there is not much use to porting the algorithm as-is to the GPU. To investigate alternatives, this chapter will focus on the construction of histograms which can subsequently be used to find estimates on the PV locations.

9.2 CPU Histogramming

For the sake of completeness, the concept of histogramming will be discussed briefly, and the algorithm to generate histograms will serve as a baseline for the development and comparison of the GPU algorithms of the subsequent sections.

A histogram is a datastructure consisting of *bins* that correspond to certain value-ranges within which data-values may lie. The value of a bin denotes the number of datapoints that fall within this range. The algorithm (again denoted in C++) to construct said structures (using a `std::vector<size_t>` as the underlying container) is as follows:

```
1 std::vector<size_t> histogram(std::vector<double> const &data,
2                               double leftBound,
3                               double rightBound,
4                               size_t nBins)
5 {
6     double const binWidth = (rightBound - leftBound) / nBins;
7
8     std::vector<size_t> ret(nBins);
9     for (double x: data)
10         ++ret[(x - leftBound) / binWidth];
11
12     return ret;
```


From the above listing, it is straightforward to determine that the algorithmic complexity is $\mathcal{O}(N)$, where N is the number of datapoints (i.e. the number of iterations of the loop). Note that the above does *not* do any boundchecks before incrementing the corresponding bin. This will lead to problems when `leftBound` and/or `rightBound` are not defined properly, i.e. when some datapoints fall outside the range `[leftBound, rightBound]`.

9.3 GPU Histogramming

The simple act of histogramming becomes much more complicated in a parallel context. First of all, a decision has to be made on whether to parallelise w.r.t. bins or data, hereafter referred to as bin-parallel and data-parallel algorithms. The former dedicates one or multiple threads to a subset of bins. Each thread then scans the entire dataset and fills only the bins that have been assigned to it. The latter is parallelized with respect to data, meaning that each thread handles only a part of the data and has the entire histogram at its disposal to bin the values.

9.3.1 Bin-Parallel Algorithms

The reason one might consider a bin-parallel algorithm is to avoid data-races on the bins. Data-races occur when multiple (parallel) threads need access to the same bin to increment its value. While an increment-operation is a single instruction to the processor, it actually involves multiple sub-instructions:

1. Fetch the current value from memory and store it in a (thread-local) register.
2. Increment the register.
3. Write the value back to memory.

When thread i and j both need to increment the same value x , it is possible that they fetch the exact same value from memory, increment it, and write it back. Instead of having become $x + 2$, the value is now $x + 1$ despite two increment operations.

These kinds of race-conditions can be prevented by using *atomic* operations, i.e. operations that cause access to the memory they operate on to be blocked while the operation is in progress. CUDA provides a set of built-in atomics that can be used to this purpose, listed in Table 9.1.

While these functions definitely prevent race-conditions and thereby guarantee the integrity of the data (when used appropriately), a performance-penalty is introduced because threads are stalled until access to the data is granted. This becomes a problem when the memory-access pattern is very dense, i.e. many different threads need access to the same data.

The bin-parallel algorithms circumvent race-conditions and therefore the use of atomic functions altogether because every thread has one bin (or multiple bins) to itself. The downside is that it involves looping over all data, which might become a problem when the dataset is large.

9.3.2 Data-Parallel Algorithms

Contrary to the bin-parallel approach of Section 9.3.1, the data-parallel algorithms share the entire histogram among all threads, which implies that race-conditions have to be dealt with carefully. This is a problem that can be overcome using atomic functions, block-synchronized accesses (using `__syncthreads()`), or combinations of these. The advantage is that, for large datasets, the (read-only) access to the data can be distributed over many different threads. Also, memory-accesses (writes) to the histogram become more and more spread out (less dense) with an increasing histogram resolution. This has a positive effect on the overhead due to use of atomic accesses.

9.3.3 Parallel Implementations

As many as 6 different implementations have been tested on performance. Many of the implementations differ only slightly to investigate the effect different approaches to the same problem. In this section, each of the implementations (4 bin-parallel, 2 data-parallel) will be briefly described with a reference to its implementation, all of which can be found in Appendix A. The performance results follow in Section 9.4.

Each of the parallel kernels have the following interface:

```
1  __global__ void histogram(double const *data, uint n, uint *hist,
2                               double leftBound, double rightBound,
3                               uint nBins)
```

Here, `data` is a pointer to the data, `n` the amount of data hiding behind this pointer, `hist` a pointer to the histogram, and the three remaining parameters describe the properties of the histogram: `leftBound`, `rightBound` and `nBins`. The caller of this kernel needs to make sure that `hist` actually points to enough allocated memory to hold `nBins` values, and that the values stored in `data` all lie between `leftBound` and `rightBound`:

Data-Parallel: Naive

The most straightforward way of computing the histogram in parallel is to have each thread evaluate a subset of the data. This is accomplished in the listing of Appendix A.1, where each thread loops over the dataset with a stride of `nThreads`, i.e. the total number of threads executing the kernel. Because different threads might need access to the same bin simultaneously, the bins have to be incremented atomically to avoid race-conditions.

Data-Parallel: Shared Memory

The problem with the previous kernel is that each of the atomic accesses are to global data. This access is very slow compared to accesses in shared memory, so this is where some performance-gain might be found. Instead of communicating directly to global memory, this evolution of the

histogramming-kernel will first construct a partial histogram in shared (block-local) memory, and then add this entire structure to the histogram in global memory.

The above method does, at least to the eye, complicate the kernel a little, since now 3 steps are required to accomplish the task:

1. Initialize the shared memory segment to zeros.
2. Populate the shared (partial) histogram.
3. Add the partial histogram to the global histogram.

Each of these steps is dependent on the previous step, so barrier-synchronizations are needed in-between. Its listing can be found in Appendix A.2.

Bin Parallel: Naive

In this initial bin-parallel implementation, every thread will populate a unique subset of bins (rather than datapoints) in global memory. For every bin, it scans the entire dataset to look for elements that fall within this range, keeping a local counter. This counter is then assigned only once to the corresponding histogram entry in global memory to minimize the number of global memory writes (which are slow). Appendix A.3 shows the implementation of this algorithm. Although this implementation lacks the need for any synchronizations or atomic operations, the repeated scanning of the entire dataset (inner loop) does seem wasteful. Attempts to improve upon this will be worked out in later versions.

Bin Parallel: Blocks

Rather than assigning a single bin to every *thread*, the next implementation assigns a single bin to every *block*. This allows the data-iteration to be parallelized on a block-level, while maintaining the bin-parallel nature on a grid-level. In addition to a counter local to every thread (as in the previous version), the threads within each block share a counter in shared memory. This shared counter is then updated atomically by each thread. When all threads have contributed to the shared counter, the global histogram is updated (non-atomically). The implementation can be found in Appendix A.4.

Bin Parallel: Warps

Another way of parallelizing the iteration of the data is on the warp-level. Every 32 threads within a block (at least in the generation of hardware at the time of writing) make up a warp, which executes instructions in lock-step (Section 5.3). Threads within a warp have a limited but efficient way of communicating directly to each other through the use of voting functions. Given some boolean predicate, each thread of a warp can set a bit (of a 32-bit word) to either 0 or 1 to indicate whether this predicate holds for this thread. The `_ballot()` built-in function returns a 32-bit integer of which the individual bits represent the outcomes of every thread.

The ballot-function can be used to check which of 32 data-points fall within the range of the currently examined bin. Its return-value must then be processed to count the number of set bits, also known as its pop(ulation)-count. CUDA has a built-in function to do just this: `--popc()`.

The kernel listed in A.5 is conceptually similar to the previous one (A.4), with the difference that each warp -rather than each block- will be given a set of bins, which they populate using the techniques above. To correct for situations where the number of threads is not an exact multiple of the warp-size, each thread has to calculate its `effectiveWarpSize`, which is the number of *active* threads in its warp (and thus the number of participating threads in the voting procedure). For example, a kernel of 70 ($= 2 * 32 + 6$) threads, a total of 3 warps is allocated. However, the third warp consists of only 6 threads.

Bin Parallel: Warps + Blocks

The next implementation (Appendix A.6) is a combination of the above where every block handles one bin at a time, but the ballot-function is used to parallelize the data-iteration on a warp-level. Again, a counter is kept in shared memory to reduce global-memory accesses.

Bin/Data Parallel: Shared Memory

The final implementation is a hybrid between the data-parallel and bin-parallel kernels. Within each block, the iteration over the dataset is parallelized to avoid having each thread looping over all datapoints (which was the case in some of the previous bin-parallel kernels). Now, rather than checking the entire dataset for each bin, each thread checks all bins (within a subset assigned to its block) for each datapoint. In the code, this is visible as a reversal of the loops: the outer loop is over data while the inner loop is over the (sub)histogram (Appendix A.7). A buffer of shared memory is kept to store the partial histogram.

9.4 Results

The performance of the algorithms described above (listed in Appendix A) greatly depends on the size of the data, the number of bins used and the thread-distribution with which the kernels are launched. To get a rough idea of the performance-differences, runtimes have been measured on *uniformly distributed* datasets for varying histogram-sizes.

Figure 9.1 shows the runtimes (relative to those of the CPU) as a function of the number of data-elements of each of the algorithms described in the previous section (lower is better). From this figure, it seems that the data-parallel algorithms outperform the bin-parallel algorithms basically across the entire range. In fact, the difference is so large that for such amounts of data, only the data-parallelized algorithms can beat the trivial sequential CPU implementation.

The shared memory optimization (red line) does not outperform the naive version until very large datasets are handled ($\sim 45,000$ elements). The theory behind the shared memory optimization was that instead of many single increments directly on global memory, a block-local

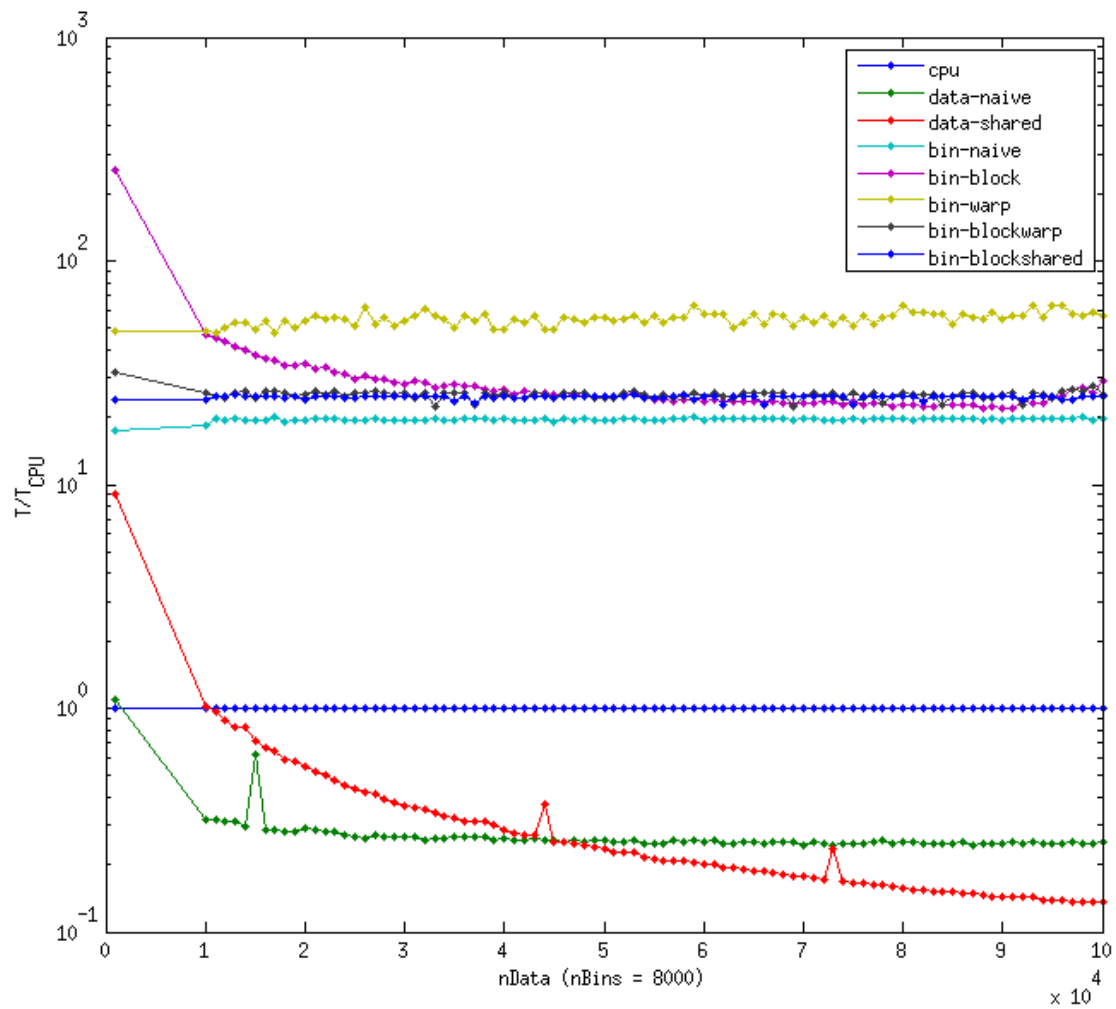


Figure 9.1: Overview of all algorithms on large datasets and large histograms. The curves have been normalized with respect to the CPU benchmarks, therefore effectively showing their speedup.

buffer is kept in shared memory, which is directly written to global memory upon completion. This reduces the number of global writes and thread-stalls. However, with as many as 8000 bins the probability of collisions is low (in a uniform distribution) and the overhead of thread-synchronizations pays off only when the dataset becomes really vast.

The two data-parallel algorithms are compared in more detail in Figure 9.2 where their performance is plotted against the number of bins involved for both a large dataset (100,000 elements) and a very small dataset (100 elements). Note that, in a sequential algorithm like that run on a CPU, the runtime would be solely dependent on the number of data-elements and would therefore be constant as a function of `nBins`. The rather complicated trends seen in Figure 9.2 can therefore be purely attributed to effects due to thread congestion.

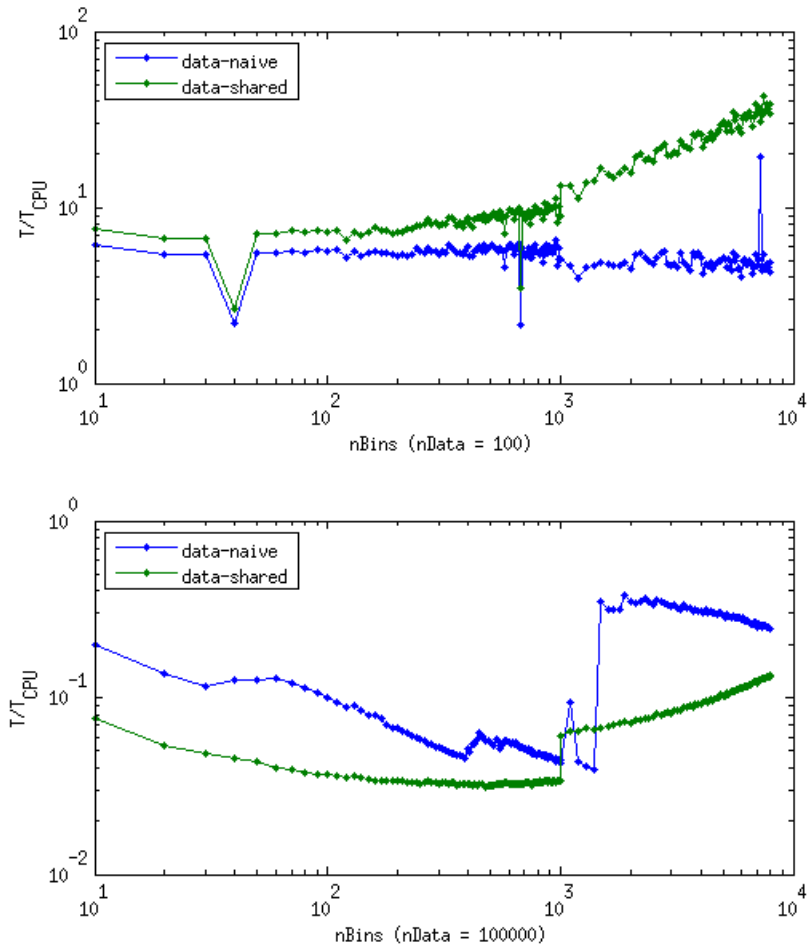


Figure 9.2: Comparison of the data-parallel histogramming algorithms for both a small and large dataset.

| Atomic Function | Description |
|----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>atomicAdd(addr, val)</code> | Add <code>val</code> to the variable at <code>addr</code> (pointer-value). Returns the old value to its caller. |
| <code>atomicSub(addr, val)</code> | Subtract <code>val</code> from the variable at <code>addr</code> (pointer-value). Returns the old value to its caller. |
| <code>atomicExch(addr, val)</code> | Exchange the value at <code>addr</code> for the new value <code>val</code> and return the old value. |
| <code>atomicMin(addr, val)</code> | Store the minimum of <code>val</code> and the value at <code>addr</code> back at <code>addr</code> and return the old value. |
| <code>atomicMax(addr, val)</code> | Store the maximum of <code>val</code> and the value at <code>addr</code> back at <code>addr</code> and return the old value. |
| <code>atomicInc(addr, val)</code> | If the value at <code>addr</code> is greater or equal to <code>val</code> , it will be incremented. If not, it will be set to 0. The old value is returned to the caller. |
| <code>atomicDec(addr, val)</code> | If the value at <code>addr</code> is less than or equal to <code>val</code> , it will be decremented. If not, it will be set to 0. The old value is returned to the caller. |
| <code>atomicCAS(addr, cmp, val)</code> | Compare And Swap: If the value at <code>addr</code> is equal to <code>cmp</code> , store <code>val</code> at <code>addr</code> . Returns the old value. |
| <code>atomicAnd(addr, val)</code> | Compute the bitwise AND of the value at <code>addr</code> and <code>val</code> , and store the result at <code>addr</code> . Returns the old value. |
| <code>atomicOr(addr, val)</code> | Compute the bitwise OR of the value at <code>addr</code> and <code>val</code> , and store the result at <code>addr</code> . Returns the old value. |
| <code>atomicXor(addr, val)</code> | Compute the bitwise XOR of the value at <code>addr</code> and <code>val</code> , and store the result at <code>addr</code> . Returns the old value. |

Table 9.1: List of CUDA built-in atomic operations on memory that is accessible to multiple threads, to avoid race-conditions to the data. For more detailed information, e.g. specific types for which these functions have been overloaded, can be found in Appendix B.12 of the CUDA Programming Guide [?].

Chapter 10

Primary Vertices from Histograms

In the Section 9, many different parallel histogramming algorithms have been discussed. Regardless of the choice, a histogram can be produced to estimate the positions of the primary vertices. The z -coordinate for which each track has its closest approach with the beam axis can be histogrammed, producing *peaks* at the PV locations. This introduces new challenges:

1. What is the optimal value of the bin-width parameter?
2. How to define a peak, and how to reliably differentiate between peaks close to one another?

These challenges are not independent. The optimal value of the bin-width depends on the definition of a peak and subsequently the peak-finding algorithm that does the search. In the remainder of this text, the most straightforward definition of a peak is adopted: *a peak in the histogram is defined as a bin that is surrounded by lower values at both sides.*

10.1 Optimal Bin-Width

In order for the peak-finding algorithm to produce reliable results, the bin-width parameter w has to be set such that enough resolution is gained to find PV that are close to each other without producing too much noise. To find a method of estimating the optimal value w_{opt} , a normal distribution $f(z) = \mathcal{N}(\mu, \sigma)$ around the *true* PV's is assumed. This section describes an attempt of finding w_{opt} by maximizing the probability that the resulting histogram, drawn from an underlying distribution that follows $f(z)$, resembles that distribution as close as possible.

For very small bins of width w , centered around some point z_i , the histogram will become noisy due to the fact that the probabilities p_i and p_{i+1} (adjacent bins), where

$$p_i = \int_{z_i - w/2}^{z_i + w/2} f(z) dz, \quad (10.1)$$

are very close to each other, i.e.

$$\lim_{w \rightarrow 0} |p_i - p_{i+1}| = 0. \quad (10.2)$$

Because of almost equal probabilities in adjacent bins, the resulting histogram bars are likely to produce peaks outside of the peak of the underlying distribution.

The goal is then to find a value for w such that the probability of finding false peaks due to statistical noise is minimized for a given number of measurements $N = \sum_i N_i$ (where N_i is the count of bin i). In other words, the probability that, if $p_i > p_{i+1}$, one will find that $N_i > N_{i+1}$ and vice versa. The probability P_i is defined as the probability of finding that the sign of the gradient between bin i to $i + 1$ is conform the underlying PDF:

$$\begin{aligned} P_i &\equiv \begin{cases} P(N_i > N_{i+1}), & p_i > p_{i+1} \\ P(N_i < N_{i+1}), & p_i < p_{i+1} \end{cases} \\ &= P(\text{sgn}(N_i - N_{i+1}) = \text{sgn}(p_i - p_{i+1})) \end{aligned} \quad (10.3)$$

This probability can be calculated by summing over every possible combination of bin-values $(k, l) \in \mathbb{N}^2$ for which these conditions (e.g. $N_i > N_{i+1}$) hold, provided that $N_i + N_{i+1} \leq N$ ¹:

$$P_i = \begin{cases} \sum_{\substack{k < l \\ k+l \leq N}} P(N_i = k) \cdot P(N_{i+1} = l), & p_i < p_{i+1} \\ \sum_{\substack{k > l \\ k+l \leq N}} P(N_i = k) \cdot P(N_{i+1} = l), & p_i > p_{i+1} \end{cases}. \quad (10.4)$$

Here, the probability that some bin takes a certain value x , $P(N_i = k)$, can be evaluated by the binomial distribution $\text{Bin}(N, p_i)$:

$$P(N_i = k) = p_i^k (1 - p_i)^{N-k} \cdot \binom{N}{k}. \quad (10.5)$$

The expectation-value of P_i , denoted

$$\langle P \rangle = \sum_i p_i P_i. \quad (10.6)$$

must be maximized with respect to w , yielding the optimal bin-width as per this definition.

10.1.1 Results

The results of this analysis depend on the underlying distribution $f(z)$. For a singly peaked standard normal distribution $f(z) = \mathcal{N}(z; 0, 1) \equiv \mathcal{N}(z)$, the statistic $\langle P \rangle$ goes to 1 as the binwidth w grows (Figure 10.1). This is to be expected because a very large bin will cover the entire data-range and will therefore produce a single large peak. However, this choice of $f(z)$

¹This is actually a simplification, because one should actually sum over all possible histograms containing N hits. This number is of course much too large for practical use, so instead we choose to sum only over all possibilities for 2 bins, limited by the total histogram count.

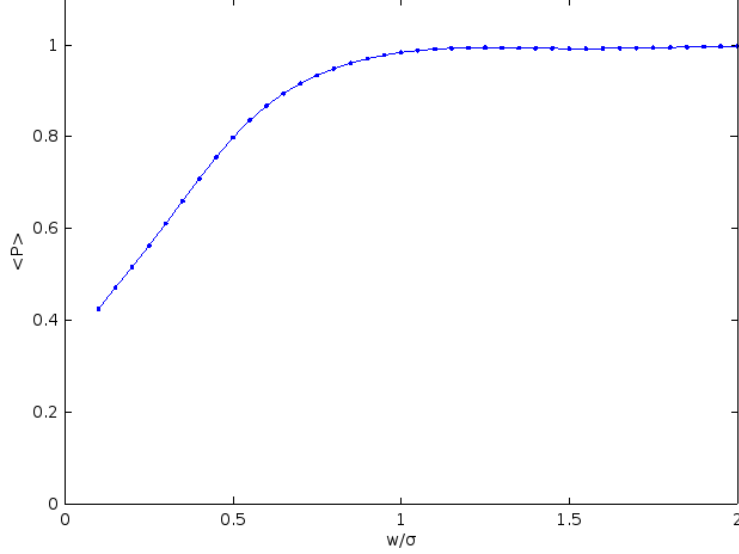


Figure 10.1: Expectation value $\langle P \rangle$ (Eq. 10.6) for a singly peaked PDF ($f(z) = \mathcal{N}(0, 1)$).

does not reflect the resolution requirement. In order to do so, the PDF is modified to include a second peak at a distance r (Figure 10.2):

$$f(z; \mu, \sigma) = \frac{1}{2} \left[\mathcal{N}(z; \mu - \frac{r}{2}, \sigma) + \mathcal{N}(z; \mu + \frac{r}{2}, \sigma) \right]. \quad (10.7)$$

Figure 10.3 shows the results of $\langle P \rangle$ (Equation 10.6) as a function of the normalized bin-width $\frac{w}{\sigma}$ for multiple values of N . As with the singly peaked distribution, the probability tends to unity for large bins. For smaller bins, local maxima have formed, the largest of which *could* be of interest to the algorithm. Figures 10.3 and 10.4 show that the position of the maximum shifts to the left and the actual height of this maximum is increasing with increasing N . This means that, not surprisingly, a larger dataset will produce a better representation of the underlying distribution and allows for higher resolution (smaller bins). To determine w_{opt} from Figure 10.3, the middle peak was determined numerically. The position of the maximum shifts to the left and the actual height of this maximum is increasing with increasing N . As expected, this means that a larger dataset will produce a better representation of the underlying distribution and allows for higher resolution (smaller bins). Because of the peaked nature of these curves, the position of the maximum w_{opt} can be plotted against N , which is shown in Figure 10.4 together with a more common method of choosing the bin-width, known as the Freedman-Diaconis rule:

$$w_{FD} = 2\text{IQR}(z)N^{-1/3}, \quad (10.8)$$

where $\text{IQR}(z)$ denotes the interquartile range of the data z , or the difference between the third and first quartile $Q_3 - Q_1$. For a normal distribution, $\text{IQR} = 1.32\sigma$, which was used in Figure 10.4 to evaluate (10.8).

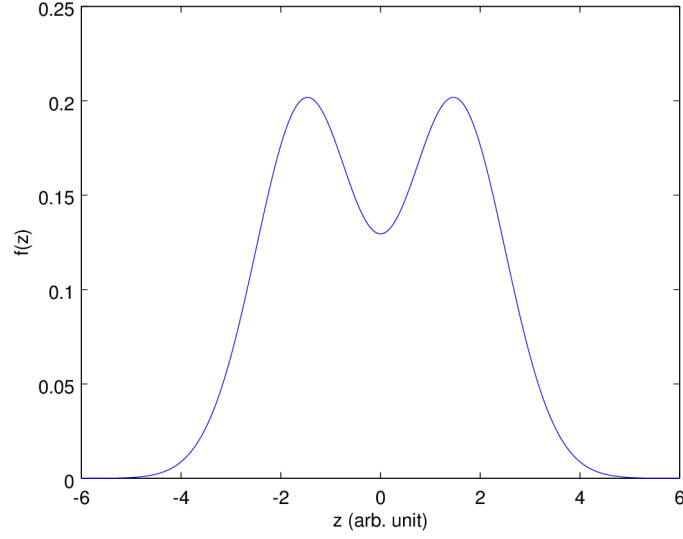


Figure 10.2: Plot of Equation 10.7 for $r = 3\sigma$ ($\mu = 0$, $\sigma = 1$).

Despite the (small) discrepancy between this analysis and the Freedman-Diaconis rule, we can conclude that in order for the peakfinding algorithm to find true peaks, the bin-width must be chosen quite wide. For typical events where $N \sim 500$, a binwidth of around 0.4σ is optimal for finding peaks at a resolution of $r = 3\sigma$. Follow-up research could explore the behavior of w_{opt} as a function of r , and check through simulation whether the predicted optimal bin-width actually yields the best results in a peak-search under the assumptions made in this chapter. We expect that for smaller r , the optimal bin-width must be chosen smaller as well, at the cost of more false positives. Larger displacements between peaks will of course allow for a more coarse choice of w .

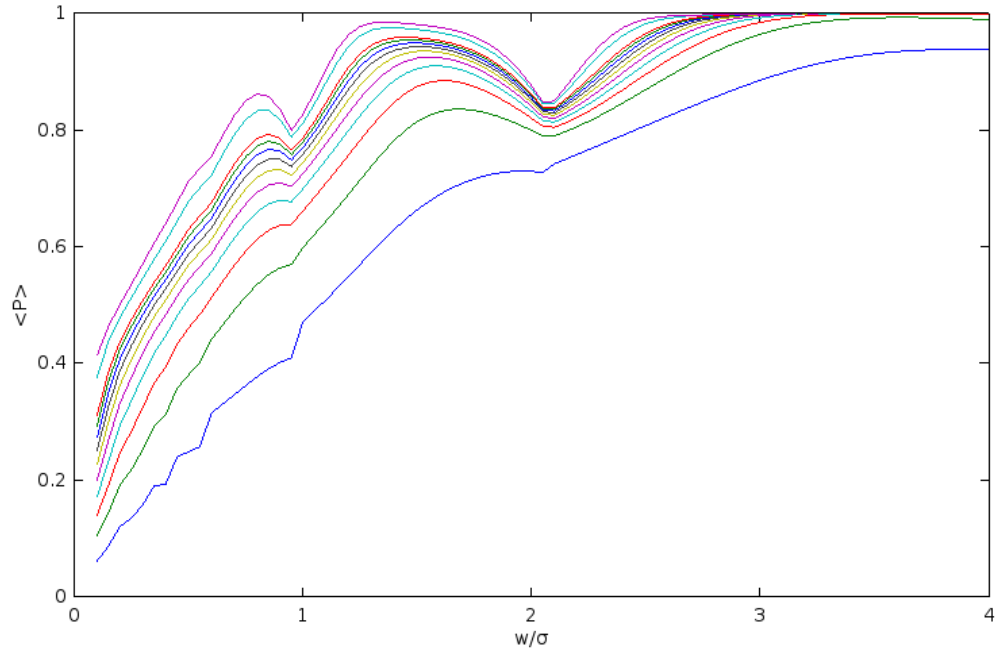


Figure 10.3: Expectation value $\langle P \rangle$ (Eq. 10.6) for a doubly peaked PDF (Eq. 10.7).

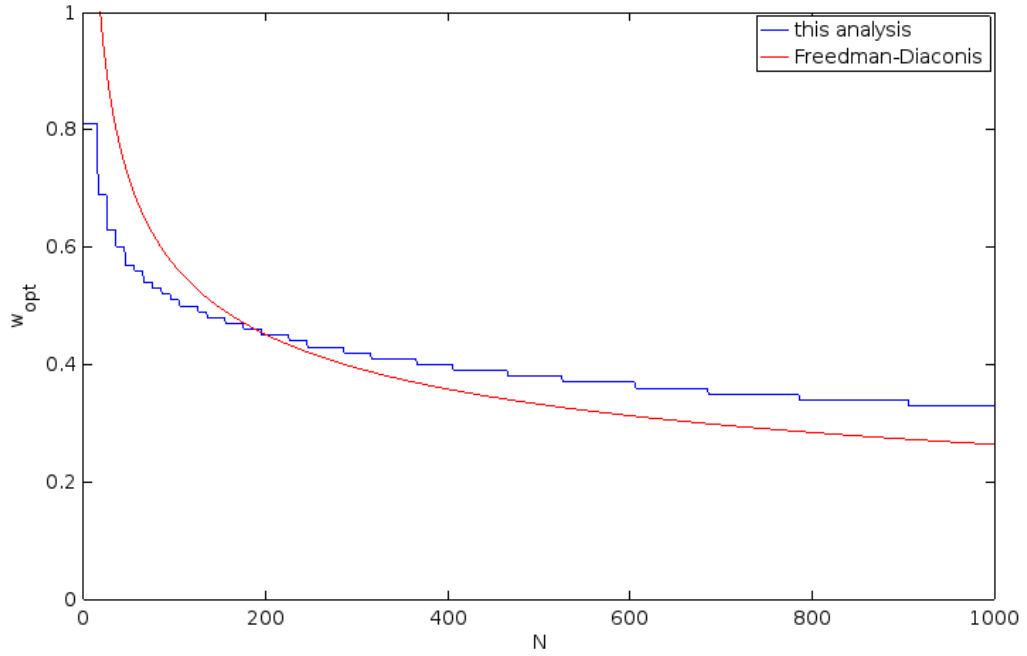


Figure 10.4: Comparison of the results of this analysis of Section 10.1 versus the Freedman-Diaconis rule (Eq. 10.8).

Chapter 11

Secondary Vertices

11.1 Candidates

So far, this analysis has concerned itself with two techniques that can be considered independent from one another. The adjacency-matrix can be constructed separately from the histogram and vice versa. To find secondary vertices, aforementioned parts need to be combined. Knowledge of the primary vertices, however acquired, will be used to ‘clean up’ the adjacency graph, leaving candidates for further analysis. Finding true secondary vertices with reasonable accuracy has proven very difficult, but the dataset *can* be reduced by a large amount using well-scaling parallel algorithms. Further and more thorough research is needed to establish whether this reduction is sufficient for conventional (off-line) algorithms to find secondaries during the on-line analysis. Forthcoming sections will (qualitatively) discuss algorithms that remove nodes from the graph which to a reasonable certainty can be identified as (uninteresting) tracks originating from primary vertices.

In Table 11.1, this amounts to minimizing b rather than maximizing d . In other words, rather than optimizing an algorithm to reliably identify secondary tracks (d), the algorithm aims to eliminate as many primary tracks as possible, whilst retaining the secondary tracks.

11.2 Origin Cut

Every track-crossing (1’s in the adjacencymatrix) could potentially be a vertex, so the goal is to eliminate as many crossings in order to keep a viable set of candidates for further analysis. The

| | | track | |
|------------------|---|-------|---|
| | | P | S |
| identified as | P | a | b |
| | S | c | d |

Table 11.1: Success-matrix, where P and S denote primary and secondary tracks respectively

origin cut looks at each adjacent pair of tracks to see where these tracks could have originated from. Most crossings are found at the primary vertex locations, but every now and then two tracks cross outside these boundaries. This could be coincidental, or it might be a secondary vertex. In the latter case, the angle of its constituents is generally small due to *beaming* (see 11.2.1). Due to this effect, secondary tracks point back to one of the primary vertices. This allows the algorithm to eliminate accidental crossings by determining the most likely origin of each tracks in a non-primary vertex and compare to each other. If they agree, the crossing is conserved. If not, it is erased from the adjacency matrix. Higher beaming rates and/or less primary vertices will increase the differentiative power of this algorithm, thereby improving the efficiency of such a cut. Figure 11.1 shows a 2D example of this process.

Alternatively, one might set a threshold on the opening angle of (a pair of) tracks originating from a suspected secondary vertex. However, the algorithm already depends on many free parameters, therefore the choice was made to avoid adding yet another degree of freedom.

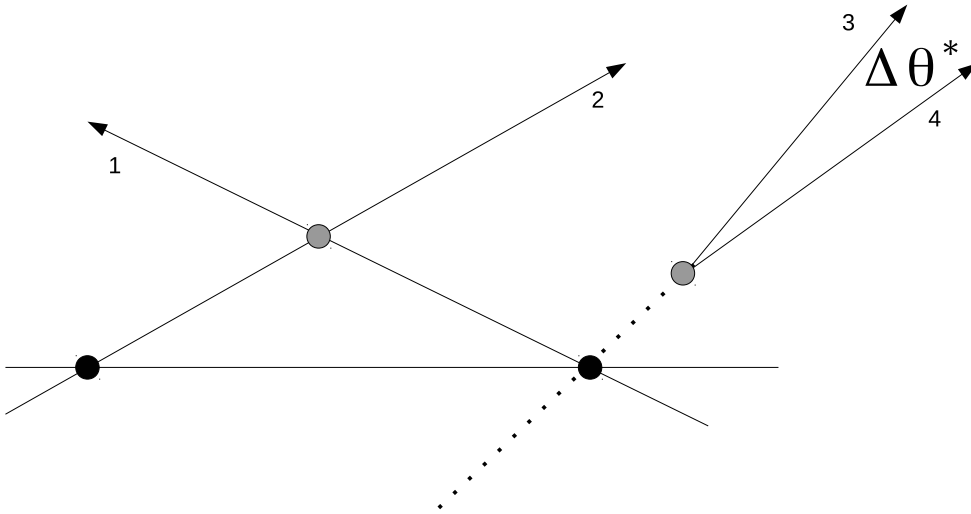


Figure 11.1: Example event where two track crossings (gray) are being examined by the origin-cut routine. Track 1 and 2 are originating from different PV's (black), therefore cannot be a valid SV candidate. Track 3 and 4 are beamed (see 11.2.1) and have a small opening angle $\Delta\theta^*$, pointing back to the same PV.

11.2.1 Beaming

Beaming is a (partly) relativistic effect that occurs when a particle decays at high velocity, resulting in the decay products having mostly forward momentum. This is important in the analysis because it allows for disregarding any vertices whose tracks point in wildly different directions. This section will consider the 2-particle case to illustrate the effect analytically.

Due to conservation of momentum, the decay products are emitted in opposite directions seen from the centre of mass frame centered on the parent particle (such that $\Delta\theta = |\theta_1 - \theta_2| = \pi$). However, in the lab-frame, this angle is much smaller. The effect thus seen in the lab system is that the particles are both emitted in the direction of their parent's momentum, yielding a relative angle much smaller than π : $\Delta\theta^* < \pi$.

Given a rest-frame centered on the center of mass of the parent-particle, the total energy of the system is denoted (in units where $c = 1$)

$$E_{tot} = E_p^{CM} = m_p. \quad (11.1)$$

This system has a boost

$$\beta = \frac{p_p^*}{E_p^*}, \quad (11.2)$$

with respect to the lab-frame (denoted by the superscripted asterix), so in order to acquire the momentum of a child-particle in the lab system, a Lorentz transformation is applied. It follows that:

$$E_c^* = \gamma (E_c^{CM} + \beta p_{\parallel}) \quad (11.3)$$

$$p_{c,\parallel}^* = \gamma (\beta E_c^{CM} + p_{c,\parallel}^{CM}) \quad (11.4)$$

$$p_{c,\perp}^* = p_{c,\perp}^{CM}, \quad (11.5)$$

where p_{\parallel} and p_{\perp} denote the parallel and perpendicular component of the particle's momentum respectively. Therefore, the direction of decay, seen from the lab-system, is (dropping the child-subscript c and superscript CM , which are now implicit to every symbol other than β)

$$\begin{aligned} \tan(\theta^*) &= \frac{p_{\perp}^*}{p_{\parallel}^*} \\ &= \frac{p_{\perp}}{\gamma (\beta E + p_{\parallel})} \\ &= \frac{p \sin \theta}{\gamma (\beta E + p \cos \theta)}. \end{aligned} \quad (11.6)$$

In a first order approximation where the angle θ^* is small, $\beta \approx 1$ and the energy of the decay particles is governed primarily by their momentum, the difference $\Delta\theta^*$ can be shown (using

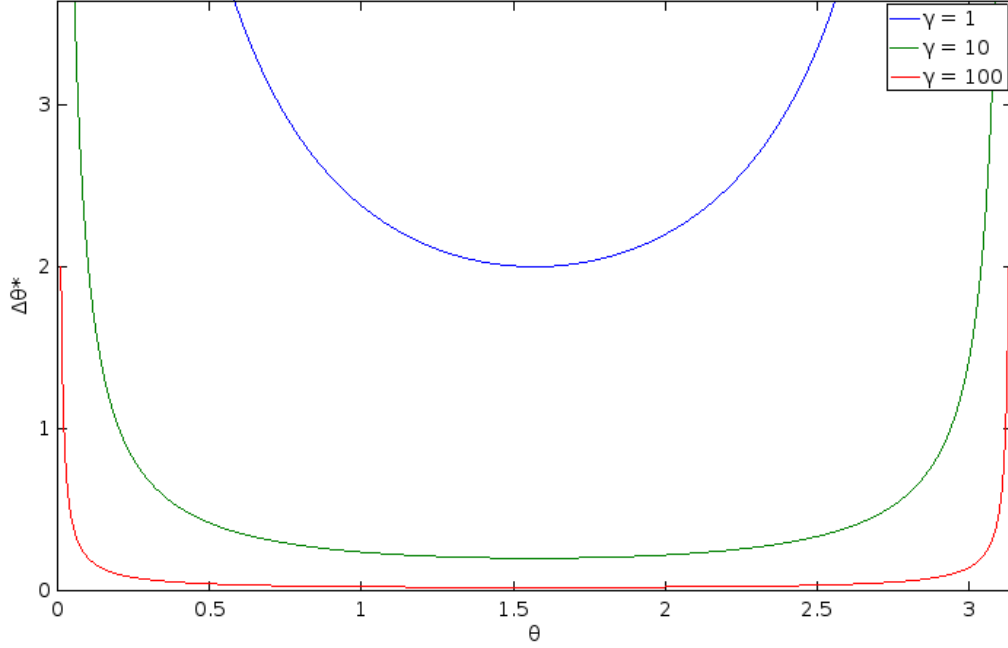


Figure 11.2: The approximated beaming effect is demonstrated for two values of γ (red: $\gamma = 100$, blue: $\gamma = 10$) by plotting the decay-angle θ versus the opening angle $\Delta\theta^*$ as seen from the lab-system. A flatter line corresponds to a larger beaming effect, which indeed occurs at higher velocities.

that $\theta_1 - \theta_2 = \pi$) to be

$$\Delta\theta^* = \arctan\left(\frac{p \sin \theta_1}{\gamma(\beta E_1 + p \cos \theta_1)}\right) - \arctan\left(\frac{p \sin \theta_2}{\gamma(\beta E_2 + p \cos \theta_2)}\right) \quad (11.7)$$

$$\begin{aligned} &\approx \frac{p \sin \theta}{\gamma p(1 + \cos \theta)} - \frac{p(-\sin \theta)}{\gamma p(1 - \cos \theta)} \\ &= \frac{2}{\gamma \sin \theta} \end{aligned} \quad (11.8)$$

Figure 11.2 shows a plot of Equation 11.8 for two typical values of γ . The beaming effect can be seen to be larger for larger boosts, as expected.

The above equation has been derived assuming massless particles, as a result of which the momentum p dropped out of the result. When no such assumption is made, p becomes a function of the masses of the parent particle and its children. Using that $p_1 = p_2 = p$ (conservation of momentum) and the conservation of energy of the stationary parent particle, expressed as

$$\sqrt{m_1^2 + p^2} + \sqrt{m_2^2 + p^2} = m_p, \quad (11.9)$$

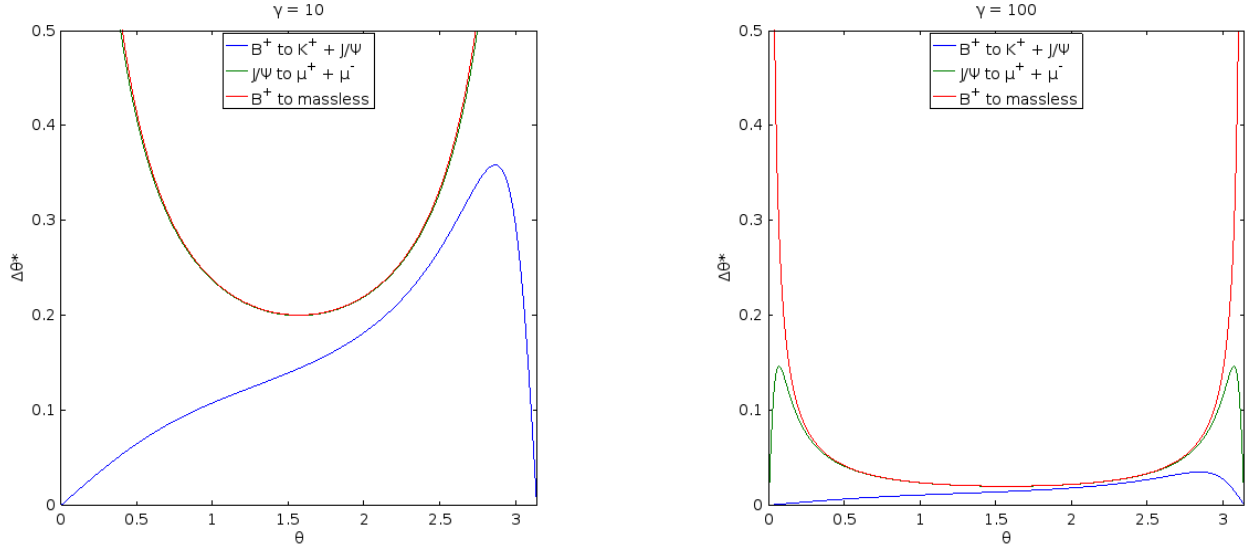


Figure 11.3: Equation 11.7 has been solved for two particular decay-channels and the massless case which has been approximated by Equation 11.8. The trivial case ($\gamma = 1$) has not been shown, as in this case $\theta^* = \theta$ and therefore $\Delta\theta^* = \pi$.

one can solve for p and find that

$$p = \frac{1}{2m_p} \sqrt{m_p^4 + m_1^4 + m_2^4 - 2[m_1^2 m_2^2 + m_p^2(m_1^2 + m_2^2)]}. \quad (11.10)$$

Using (11.10), Equation 11.7 can be solved exactly depending on the decay channel. The solutions for $B^+ \rightarrow J/\psi + K^+$ and $J/\psi \rightarrow \mu^+ \mu^-$ are shown in Figure 11.3.

11.3 Radial Cut

Another constraint on the topology of each event is the fact that secondary vertices are displaced from the primary vertices by the decay length of the parent particle

$$\lambda = \beta c \tau \gamma, \quad (11.11)$$

where τ is the mean lifetime of the parent particle. For example, the average decay length of a B^+ ($\tau = 1.64 \times 10^{-12} \text{ s}$) lies in the order of centimeters (6-60 mm). In other words, crossings (nonzero's in the adjacency matrix) can be ruled out as SV candidates when they lie too close to a PV.

Due to uncertainties in the track-parameters, tracks will not intersect perfectly in the PV (Figure 11.4). However, given accurate enough PV positions along the z -axis and their corresponding errors, the so-called radial cut transforms the adjacency matrix such that crossings within a threshold radius (related to the error) from each PV are set to zero, leaving only candidates sufficiently far away from the possible origins.

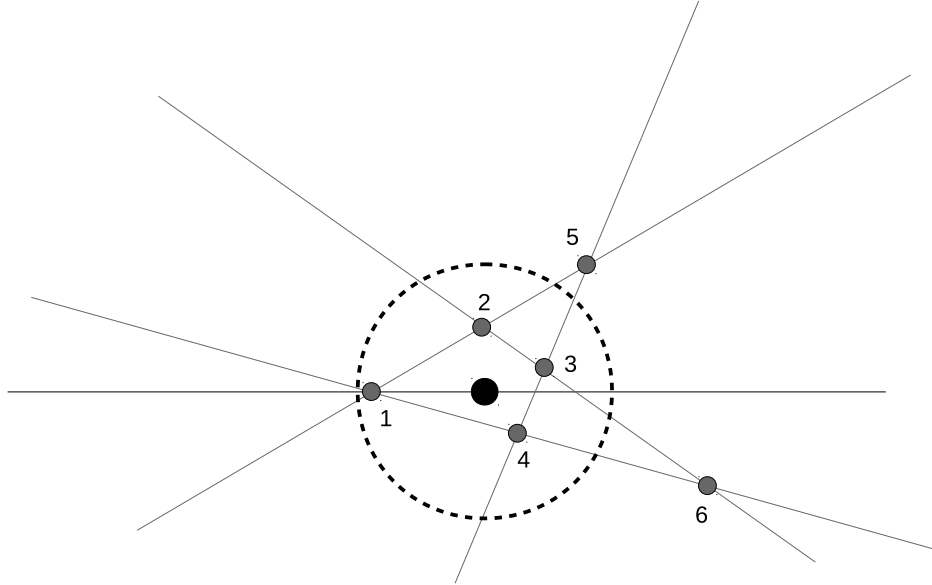


Figure 11.4: Due to uncertainties in the track parameters, many primary crossings will occur displaced from the beam axis (2, 3 and 4). Given a known error on the z -coördinate of the PV, an error sphere is constructed around it, within which all crossings can be identified as primary and therefore stop being considered as SV candidates, i.e. removed from the adjacency matrix. In this case, only crossing 5 and 6 will remain.

11.3.1 Cylinder Cut

A faster, yet less discriminative way of implementing knowledge of the decay lengths of the channel of interest would be to perform a cut independent of the PV positions. Rather than drawing a sphere around each PV, one could draw a cylinder over the entire beam-axis. This method would occasionally miss SV's that happen to have formed from parent particles that traveled further along the beam-axis. For example, in Figure 11.4, vertex 6 would have been discarded when using the same threshold-radius as used for the radial cut.

11.4 Connected Components

After all operations on the adjacency matrix have been performed, a final algorithm has to find the so-called connected components of the corresponding graph. A connected component is a subset of connected vertices that is disconnected from all other vertices (outside this subset) of the same graph. The remaining connected components are the final SV candidates, where the

number of nodes is the number of tracks involved in the suspected secondary decay.

A parallel search for connected components is inherently complicated, as the most straightforward way is a linear tree traversal (either breadth- or depth-first). During the research for this report, a parallel search has been developed, but did not perform consistent enough for the results to be included here. Further research could make use of parallel methods that transcend the scope of this report in [10], [5], [13] and [14].

11.5 Example

Figure 11.5 shows a highly simplified event that will serve as an illustration on how the previously described transformations will work together to reduce the number of SV candidates. In this figure, 7 tracks are shown in a 2D plane according to their reconstructed (direction agnostic) track parameters. Track 6 and 7 are drawn in such a way as to avoid any confusion on intersections with other tracks: for the purpose of illustration, these tracks only cross each other. The blue and red tracks, although unknown to the algorithm, have originated from the blue and red PV's respectively, whereas the black tracks originate from an SV.

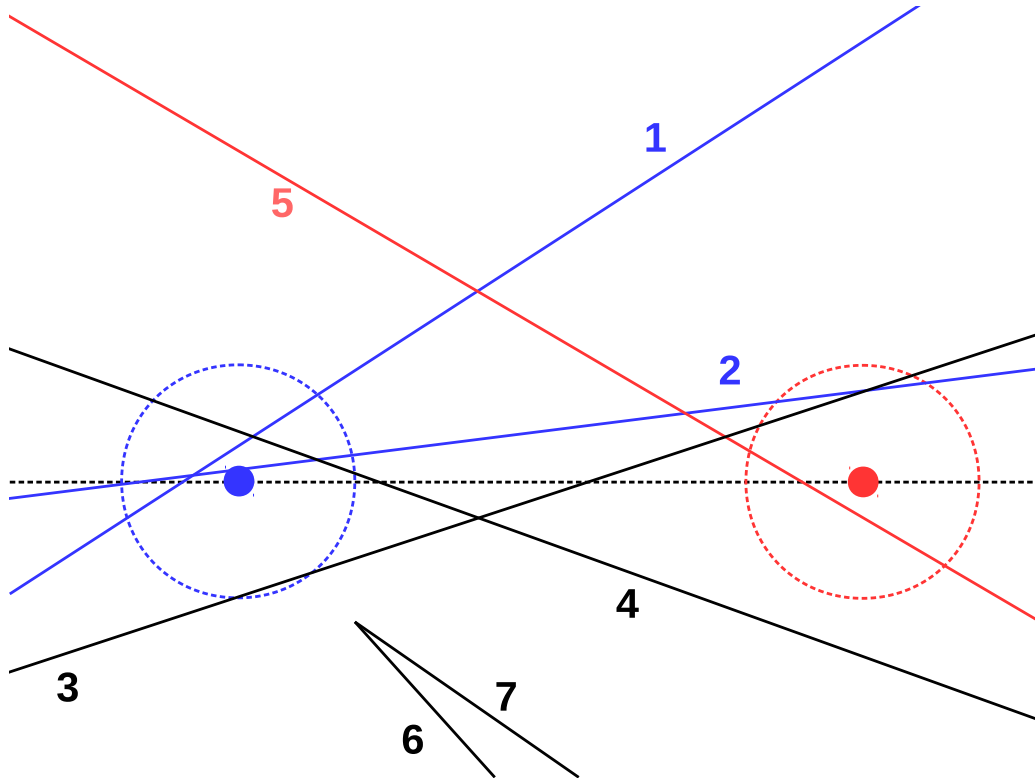


Figure 11.5: Simplified 2D projection of an event with 2 PV's and a single SV, color-coded to indicate the origin of each track.

The corresponding adjacency matrix will be represented by Figure 11.6 and subsequent figures,

where each black field at index (i, j) indicates that, at that point during the algorithm, the vertex candidate (i, j) is still under consideration.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|------|-------|-------|-------|-------|-------|-------|
| 1 | Gray | Black | White | Black | Black | White | White |
| 2 | | Gray | Black | Black | Black | White | White |
| 3 | | | Gray | Black | Black | White | White |
| 4 | | | | Gray | White | White | White |
| 5 | | | | | Gray | White | White |
| 6 | | | | | | Gray | Black |
| 7 | | | | | | | Gray |

Figure 11.6: Initial adjacency matrix of the event shown in Figure 11.5. Black squares indicate 1's in the matrix, meaning that these crossings are not (yet) eliminated by the cutting procedures.

11.5.1 Origin Cut

The origin cut removes all entries for which the crossing tracks originate from different PV's. Intersections involving tracks that haven't been assigned an origin (black) will always be preserved. In this example, only the red and blue tracks have been assigned an origin, so only crossings between these tracks can be eliminated. Therefore, $(1, 5)$ and $(2, 5)$ will be removed from the adjacency matrix, which appears as red fills in Figure 11.7. The 'O's indicate that these entries have been removed as a result of the origin cut.

11.5.2 Radial Cut

The radial cut removes all entries in the adjacency matrix that correspond to crossings that occur sufficiently close to a PV, i.e. within the indicated circular areas. In the case of Figure 11.5, crossings $(1, 2)$, $(1, 4)$, $(2, 3)$ and $(2, 4)$ will be eliminated. Figure 11.8 shows the resulting

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|------|-------|-------|-------|-------|-------|-------|
| 1 | gray | black | white | black | red O | white | white |
| 2 | | gray | black | black | red O | white | white |
| 3 | | | gray | black | black | white | white |
| 4 | | | | gray | white | white | white |
| 5 | | | | | gray | white | white |
| 6 | | | | | | gray | black |
| 7 | | | | | | | gray |

Figure 11.7: Adjacency matrix of the event shown in Figure 11.5 after the origin-cut has been applied: crossings $\{1, 5\}$ and $\{2, 5\}$ have been eliminated.

matrix, where the entries deleted are shown in red, the big ‘R’ indicating that they were deleted by the origin cut.

11.5.3 Connected Components

In the resulting adjacency matrix (Figure 11.8), two connected components are left: $\{3, 4, 5\}$ and $\{6, 7\}$. The latter we know to be a SV and the former we know to contain the other ($\{4, 5\}$).

11.6 Results

Even though the previous example is far from realistic, it does show that the number of SV candidates can be drastically reduced through these measures. At the moment of writing, both a CPU and GPU version of these algorithms has been implemented. However, neither the current quality of the implementations, nor the simulated events allow for definite quantitative statements about the performance of these algorithms. However, preliminary tests suggest that the problem size can be reduced by several orders of magnitude, allowing for more sophisticated but slower processes to be utilized for further processing.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | | R | | R | O | | |
| 2 | | | R | R | O | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |

Figure 11.8: Adjacency matrix of the event shown in Figure 11.5 after both the origin-cut and radial cut have been applied.

Chapter 12

Conclusion

As was discussed in Sections 3 and 4, the current method of finding is organized linearly and at the time of writing does not scale well to the future, where online analysis of secondary vertices is needed in conjunction with higher collision-rates. At its start, this project aimed at finding ways of accelerating this process using GPU technology, but in the process has taken multiple steps back. The attempts to develop new algorithms and software aimed at analyzing real LHCb-events have left the authors with many insights and ideas on how to proceed. There is no uncertainty whatsoever in the power of GPU technology in the scientific field. GPU's can be and are used to accelerate data analysis, even already in the Panda project (anti-Proton ANnihilation at DArmstadt), where it is used to accelerate tracking. However, their use comes with many caveats among which is a steep learning curve. While initial and simplified projects can be fun and easy to deal with, more serious projects require experience to be dealt with. What follows below is a summary of the most important experiences and prospects that were gained during the research that led to this report with regard to LHCb.

12.1 Histogramming

A key part of the vertexing pipeline that was developed in this report was a histogram to find the PV locations along the beam axis. Histogramming is trivial, both conceptually and implementation-wise, on traditional hardware, but proved to be really challenging when performed on a distributed memory system like the GPU. Individual units in such a system need to reduce the entire dataset for each of the bins in the histogram, requiring many synchronizations. Furthermore, the problem of histogramming is ‘*only*’ $\mathcal{O}(n)$. In the case of vertexing at LHCb, n is equal to the number of tracks which is of order 100-1000. These numbers are (currently) simply too small for the GPU to gain advantage over a CPU at the same task. This is exactly what was seen in Section 9, where benchmarks showed that the GPU only outperformed the CPU for datasets larger than will be encountered in real-life.

The hardware used in this research had a so-called Compute Capability 2.0. This is a measure of available features on CUDA hardware and has since far surpassed this value, introducing new

features to more recent products. One of these new features is called the *warp shuffle*, available in version 3.0 and higher, and is of great value to efficient histogramming. It allows values to be passed directly between warp-lanes, which in some cases (like histogramming) eliminates the need for atomic operations in shared or global memory. More on this can be read in one of the Parallel Forall blogs [9].

12.2 Graph Cuts

It has proven very difficult to get definitive answers to questions aimed at the performance of certain graph-cuts (entry-deletions in the adjacency graph) with respect to secondary vertex yield. Attempts have been made to do this, but the algorithms were too premature to give these answers quantitatively. Initial measurements, though not absorbed in this report, seemed to show that the expectations were holding true. That is to say that these operations, which were designed with parallelism in mind, were performing and scaling well. The following section discusses the outlook gained from this somewhat lacking conclusion.

12.3 Outlook

As noted in the previous section, definitive conclusions on the efficiency and performance of the graph-approach are still lacking. The main conclusion is then that the overall approach of starting from scratch has resulted in many insights and lots of experience, but less conclusive and definitive results. However, there is still a large potential in using GPU's to accelerate the current software pipeline. A promising method would be to find out where exactly in this pipeline the GPU could be wielded. Each stage in the current software setup should be investigated for parallel opportunities, rather than rethinking the trigger as a whole. Drop-in replacements of very specific parts could then be implemented and investigated. This has multiple advantages over the approach taken in this report:

- No need to invent new algorithms. The currently used algorithms already work, so the focus can be solely on *speed*.
- Ease of validation. When using drop-in replacements, a single event should give identical results, no matter which subroutines have been used. If the results differ after using a replacement, this means there is an error somewhere in the new implementation. This can be identified quickly and reliably.

Appendix A

Histogram Implementations

A.1 Data-Parallel: Naive

```
1  __global__ void histogram(double const *data, uint n, uint *hist,
2                             double leftBound, double rightBound,
3                             uint nBins)
4  {
5      uint const nThreads = blockDim.x * gridDim.x;
6      uint const tid = threadIdx.x + blockIdx.x * blockDim.x;
7
8      double const binWidth = (rightBound - leftBound) / nBins;
9      for (uint idx = tid; idx < n; idx += nThreads)
10     {
11         uint bin = (data[idx] - leftBound) / binWidth;
12         atomicAdd(hist + bin, 1);
13     }
14 }
```

A.2 Data Parallel: Shared Memory

```
1  __global__ void histogram(double const *data, uint n, uint *hist,
2                             double leftBound, double rightBound,
3                             uint nBins)
4  {
5      extern __shared__ uint sharedHist[];
6
7      uint const tid = threadIdx.x + blockIdx.x * blockDim.x;
8      uint const nThreads = blockDim.x * gridDim.x;
9  }
```

```

10 // Initialize shared histogram to 0's
11 for (uint idx = threadIdx.x; idx < nBins; idx += blockDim.x)
12     sharedHist[idx] = 0;
13 __syncthreads();
14
15 // Populate the shared histogram
16 double const binWidth = (rightBound - leftBound) / nBins;
17 for (uint idx = tid; idx < n; idx += nThreads)
18 {
19     uint bin = (data[idx] - leftBound) / binWidth;
20     atomicAdd(sharedHist + bin, 1);
21 }
22 __syncthreads();
23
24 // Compute resulting total (global) histogram
25 for (uint idx = threadIdx.x; idx < nBins; idx += blockDim.x)
26     atomicAdd(hist + idx, sharedHist[idx]);
27 }

```

A.3 Bin Parallel: Naive

```

1  __global__ void histogram(double const *data, uint n, uint *hist,
2                          double leftBound, double rightBound,
3                          uint nBins)
4  {
5      uint const tid = threadIdx.x + blockIdx.x * blockDim.x;
6      uint const nThreads = blockDim.x * gridDim.x;
7      double const binWidth = (rightBound - leftBound) / nBins;
8
9      for (uint bin = tid; bin < nBins; bin += nThreads)
10     {
11         double left = leftBound + (bin * binWidth);
12         double right = left + binWidth;
13
14         uint count = 0;
15         for (uint idx = 0; idx != n; ++idx)
16         {
17             double x = data[idx];
18             if (x >= left && x < right)
19                 ++count;
20         }
21
22         hist[bin] = count;

```

```
23     }
24 }
```

A.4 Bin Parallel: Blocks

```
1  __global__ void histogram(double const *data, uint n, uint *hist,
2                             double leftBound, double rightBound,
3                             uint nBins)
4  {
5      __shared__ uint sharedCount;
6
7      if (threadIdx.x == 0)
8          sharedCount = 0;
9      __syncthreads();
10
11     double const binWidth = (rightBound - leftBound) / nBins;
12     for (bin = blockIdx.x; bin < nBins; bin += gridDim.x)
13     {
14         double left = leftBound + (bin * binWidth);
15         double right = left + binWidth;
16
17         uint localCount = 0;
18         for (uint idx = threadIdx.x; idx < n; idx += blockDim.x)
19         {
20             double x = data[idx];
21             if (x >= left && x < right)
22                 ++localCount;
23         }
24         atomicAdd(&sharedCount, localCount);
25         __syncthreads();
26
27         if (threadIdx.x == 0)
28         {
29             hist[bin] = sharedCount;
30             sharedCount = 0;
31         }
32         __syncthreads();
33     }
34 }
```

A.5 Bin Parallel: Warps

```
1  __global__ void histogram(double const *data, uint n, uint *hist,
2                          double leftBound, double rightBound,
3                          uint nBins)
4  {
5      uint const warpsPerBlock = (blockDim.x + warpSize - 1) / warpSize;
6      uint const nWarps = gridDim.x * warpsPerBlock;
7      uint const warpIdx = blockIdx.x * warpsPerBlock +
8                          threadIdx.x / warpSize;
9      uint const laneIdx = threadIdx.x % warpSize;
10
11     uint effectiveWarpSize = warpSize;
12     if (blockDim.x % warpSize && 1 + threadIdx.x / warpSize ==
13         warpsPerBlock)
14         effectiveWarpSize = blockDim.x % warpSize;
15
16     double const binWidth = (rightBound - leftBound) / nBins;
17     for (uint bin = warpIdx; bin < nBins; bin += nWarps)
18     {
19         double left = leftBound + (bin * binWidth);
20         double right = left + binWidth;
21
22         for (uint idx = laneIdx; idx < n; idx += effectiveWarpSize)
23         {
24             double x = data[idx];
25             uint count = __popc(__ballot(x >= left && x < right));
26             if (laneIdx == 0)
27                 hist[bin] += count;
28         }
29     }
```

A.6 Bin Parallel: Warps + Blocks

```
1  __global__ void histogram(double const *data, uint n, uint *hist,
2                          double leftBound, double rightBound,
3                          uint nBins)
4  {
5      __shared__ uint sharedCount;
6      if (threadIdx.x == 0)
7          sharedCount = 0;
```

```

8  __syncthreads();
9
10 uint const laneIdx = threadIdx.x % warpSize;
11 double const binWidth = (rightBound - leftBound) / nBins;
12
13 for (uint bin = blockIdx.x; bin < nBins; bin += gridDim.x)
14 {
15     double left = leftBound + (bin * binWidth);
16     double right = left + binWidth;
17
18     uint localCount = 0;
19     for (uint idx = threadIdx.x; idx < n; idx += blockDim.x)
20     {
21         double x = data[idx];
22         localCount += __popc(__ballot(x >= left && x < right));
23     }
24
25     if (laneIdx == 0)
26         atomicAdd(&sharedCount, localCount);
27     __syncthreads();
28
29     if (threadIdx.x == 0)
30     {
31         hist[myBin] = sharedCount;
32         sharedCount = 0;
33     }
34     __syncthreads();
35 }
36 }

```

A.7 Bin/Data Parallel: Shared Memory

```

1  __global__ void histogram(double const *data, uint n, uint *hist,
2                          double leftBound, double rightBound, uint
3                          nBins)
4  {
5      extern __shared__ uint sharedCount[]; // enough to store all sub-
6      bins
7
8      double const binWidth = (rightBound - leftBound) / nBins;
9      uint const nSubBins = (nBins + blockDim.x - 1) / blockDim.x; //
10     must be >= shared size

```

```

9   for (uint idx = threadIdx.x; idx < nSubBins; idx += blockDim.x)
10       sharedCount[threadIdx.x] = 0;
11   __syncthreads();
12
13   for (uint idx = threadIdx.x; idx < n; idx += blockDim.x)
14   {
15       double const x = data[idx];
16       for (uint idx = 0, bin = blockIdx.x; idx != nSubBins; ++idx,
17           bin += gridDim.x)
18       {
19           double left = leftBound + (bin * binWidth);
20           double right = left + binWidth;
21
22           if (x >= left && x < right)
23               atomicAdd(&sharedCount[idx], 1);
24       }
25   }
26   __syncthreads();
27
28   for (uint idx = threadIdx.x; idx < nSubBins; idx += blockDim.x)
29       atomicAdd(&hist[blockIdx.x + (idx * gridDim.x)], sharedCount[
        idx]);

```

Bibliography

- [1] R Aaij, J Albrecht, F Alessio, S Amato, E Aslanides, et al. The LHCb Trigger and its Performance in 2011. *JINST*, 8:P04022, 2013.
- [2] J Albrecht and U Uwer. *Fast Track Reconstruction for the High Level Trigger of the LHCb Experiment*. PhD thesis, Heidelberg U., 2009. Presented on 08 Jul 2009.
- [3] Jr. Alves, A. Augusto et al. The LHCb Detector at the LHC. *JINST*, 3:S08005, 2008.
- [4] L G Cardoso, C Gaspar, O Callot, J Closier, N Neufeld, M Frank, B Jost, P Charpentier, and G Liu. Offline processing in the online computer farm. *Journal of Physics: Conference Series*, 396(3):032052, 2012.
- [5] D.V. Sarwate D. S. Hirschberg, A. K. Chandra. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.
- [6] T. Head. The lhcb trigger system. *Journal of Instrumentation*, 9(09):C09015, 2014.
- [7] Marcin Kucharczyk, Piotr Morawski, and Mariusz Witek. Primary Vertex Reconstruction at LHCb. Technical Report LHCb-PUB-2014-044. CERN-LHCb-PUB-2014-044, CERN, Geneva, Sep 2014.
- [8] Collaboration LhCb. LHCb Trigger and Online Upgrade Technical Design Report. Technical Report CERN-LHCC-2014-016. LHCb-TDR-016, CERN, Geneva, May 2014.
- [9] Justin Luitjens. Faster parallel reductions on kepler. <https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler>.
- [10] R. McColl, O. Green, and D. A. Bader. A new parallel algorithm for connected components in dynamic graphs. pages 246–255, Dec 2013.
- [11] Nvidia. *Cuda Programming Guide*. Up to date: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>.
- [12] R.M. van der Eijk. *Track reconstruction in the LHCb experiment*. PhD thesis, Universiteit van Amsterdam, 2002.
- [13] S. Even Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28:1–4, 1981.

- [14] U. Vishkin Y. Shiloach. An $o(\log n)$ prallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.