Porting the z-finder algorithm to GPU

Chris Jones

August 26, 2010

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2010

Abstract

The z-finder algorithm, a high-throughput particle physics code, was ported to GPU using C for CUDA. The algorithm, which runs in Level 2 of the ATLAS trigger, calculates the z-coordinate of a particle collision within the detector. The performance was tested on both Tesla and Fermi architectures.

It was found not to be possible to speed-up the algorithm for low luminosity samples. With high luminosity samples, a speed-up of up to 12x was attained without CUDA streams. With streams enabled, speed-up increased to a maximum of 35x, though 20x was more typical. These results suggest that it is worth investing further effort investigating porting areas of ATLAS trigger code onto GPU.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.1.1 ATLAS . . . 1
1.2 Motivation . . . 2

2 GPGPU **4**
2.1 The CUDA architecture . . . 4
2.1.1 Processing cores and multiprocessors . . . 5
2.1.2 Kernels . . . 5
2.1.3 Thread hierarchy . . . 7
2.1.4 Memory architecture . . . 9

3 Z-finder Algorithm **13**
3.1 Using triplets of spacepoints . . . 14
3.2 Serial code implementation . . . 16

4 Implementation and Benchmarking **18**
4.1 Implementation . . . 18
4.2 Benchmarking . . . 19
4.2.1 Machines . . . 20
4.2.2 Input data . . . 20

5 FindZ Kernel **21**
5.1 Floating-point precision . . . 21
5.2 Parallelism between thread blocks . . . 22
5.3 Parallelism between threads in each block . . . 22
5.3.1 Constructing histograms in parallel . . . 23
5.3.2 Calculating pairs of spacepoints . . . 25
5.4 Kernels . . . 26
5.4.1 Histogram construction in shared memory . . . 27
5.4.2 Optimising initialisation code . . . 27
5.4.3 Calculating more pairs in parallel . . . 30
5.4.4 Parallelising all pairs in slice . . . 31
5.4.5 Using texture memory . . . 32
5.4.6 Triplet mode . . . 32

6 Histogram Summation Kernel **35**
6.1 Measuring performance . . . 35
6.2 Kernels . . . 36
6.2.1 First attempt . . . 36
6.2.2 Performing summation in shared memory . . . 37
6.2.3 Coalescing memory accesses . . . 39
6.2.4 Multi-step summation . . . 42
6.2.5 Increasing the number of thread blocks . . . 45
6.2.6 Performing summation in registers . . . 46
6.2.7 Directly calculating histogram pointers . . . 47
6.2.8 Minor optimisations . . . 48
6.2.9 Final kernel . . . 48

7 Wrapper Code **50**
7.1 Pre-allocating arrays . . . 50
7.2 Consolidating spacepoint data into single array . . . 50
7.3 Overlapping CPU code and GPU code . . . 51
7.3.1 Streaming strategy . . . 51

8 Discussion **54**
8.1 Future work . . . 54
8.1.1 Load only two slices per block . . . 56
8.1.2 Using CUDA vector types . . . 56
8.1.3 Doing more work on GPU . . . 56
8.1.4 Doing more work on CPU . . . 57

9 Conclusions **58**

List of Tables

2.1 Summary of memory architecture on the two GPU devices. 12

4.1 Specifications of the two benchmarking machines. Data from CUDA SDK's deviceQuery program and [3]. 20

4.2 Properties of the two input data files, with mean averaged over the RoIs. 20

5.1 The error in the vertex when using different versions of the code. 22

List of Figures

1.1 Schematic of the trigger, showing throughput at each level. 2
1.2 The ATLAS detector. The inner detector is coloured yellow. Image taken from [2]. 2

2.1 Diagram of Fermi hardware. Image from [7]. 6
2.2 The CUDA thread hierarchy. 8

3.1 Schematic showing how z-vertices are calculated using pairs of particles within a ϕ slice. 14
3.2 The z-histogram for a low luminosity sample. Image from [9]. 15
3.3 The z-histogram for a high luminosity sample. Image from [9]. 15
3.4 The z-histogram for a high luminosity sample, using triplets of spacepoints. Image from [9]. 16
3.5 Execution time for serial code. 17

5.1 Execution time of single thread GPU code and initial thread parallel kernels. 26
5.2 Execution time of findZ kernel with histograms in shared memory. . . . 27
5.3 Execution times of the initialisation section of the findZ kernel. 28
5.4 Schematic showing how each pair of spacepoints is numbered. 30
5.5 Execution times of the findZ kernel performing more work in parallel. . 31
5.6 Execution time of findZ kernel when accessing global memory via textures references. 33
5.7 Execution times with high luminosity input after adding triplet code. . . 33

6.1 Execution times of the histogram summation and memory transfer, with summation performed on CPU and GPUs. 37
6.2 Execution time of sumHistos kernel utilising shared memory. 38
6.3 Schematic showing the alignment of memory accesses with different kernels. 39
6.4 Execution time for histogram summation with different methods of coalescing global memory accesses. 40
6.5 Execution time of sumHistos kernel with improvements to pitched array summation. 42
6.6 Schematic showing how a two-stage reduction is performed on 16 histograms. 44

6.7 Execution time of summation time when performed in multiple steps. . 44
6.8 Execution time of summation performed on a 2D grid, with different numbers of threads per block. 46
6.9 Execution time of summation after applying each of the final three optimisations applied. 47
6.10 Final execution time of histogram summation and memory transfer, with summation performed on CPU and GPU. 49

7.1 Schematic showing the progression of the calculation performed with two streams. Times for each part are very approximate. 53

8.1 Final execution time for non-overlapping GPU code compared to original CPU code. 55
8.2 Final execution time for overlapping GPU code compared to original CPU code. 55
8.3 Final execution time for GPU code compared to original CPU code, for the high luminosity sample with triplet mode enabled. 56

Acknowledgements

I would like to thank my supervisors Michele Weiland and Andrew Washbrook for their invaluable assistance throughout this dissertation. My thanks also goes to my parents for their enduring support and for giving me the opportunity to study this MSc.

Chapter 1

Introduction

1.1 Background

1.1.1 ATLAS

ATLAS (A Toroidal LHC ApparatuS) [4] is a particle detector located at CERN (European Organisation for Nuclear Research) near Geneva, Switzerland. It is one of two general-purpose detectors in the Large Hadron Collider (LHC), contributing to the search for new fundamental physics.

The main focus of the experiment is the search for the Higgs boson, often referred to as the God particle and of central importance to the Standard Model of particle physics. The Higgs is the only particle of the Standard Model that remains undetected. Potential also exists for the discovery of physics beyond the Standard Model, such as the detection of supersymmetric particles or progress towards a Grand Unified Theory; bringing the electromagnetic, strong and weak interactions under a single theory.

The ATLAS trigger

Protons travel around the collider in bunches of around 10^{11} particles. These bunches cross inside the detector at a rate of 40 MHz, with around 20 collisions per crossing [1, 10]. Around a billion collisions therefore occur every second. With 70 TB/s of data output by the detector, it is impossible to store all of it. The vast majority of physical interactions will have been observed before or are not scientifically interesting.

The purpose of the trigger is to select out the portion of the events of potential interest, reducing the amount of data down to a level where it can be stored for future analysis.

The trigger operates in three stages, each of which rejects a large portion of the events fed into it, as seen in fig. 1.1. Overall, the total number of results is reduced by a factor of around 200,000. By necessity, Level 1 of the trigger is performed on bespoke

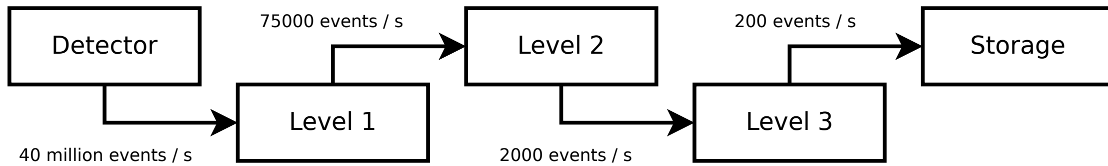

Figure 1.1: Schematic of the trigger, showing throughput at each level.

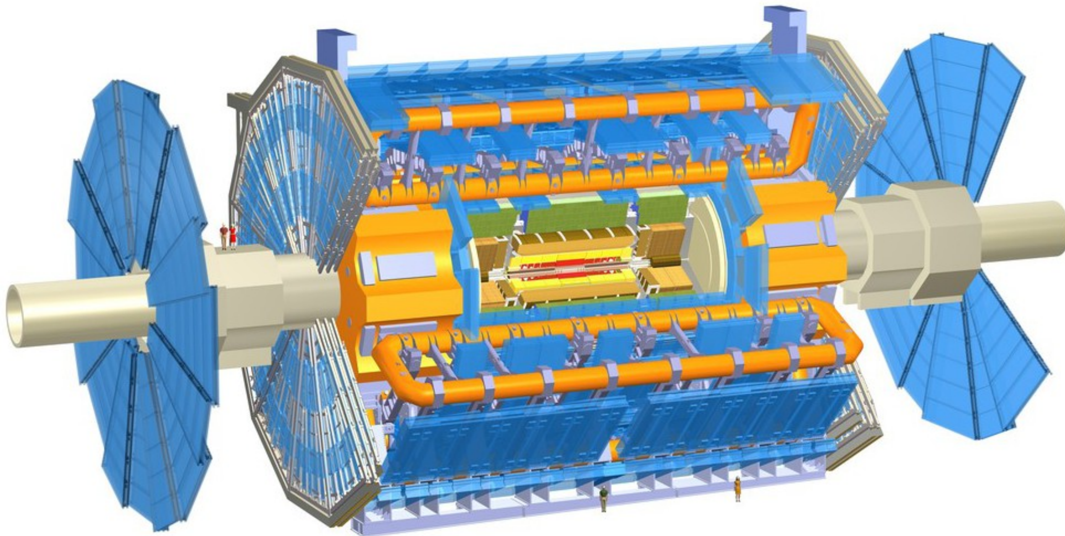

Figure 1.2: The ATLAS detector. The inner detector is coloured yellow. Image taken from [2].

hardware. Levels 2 and 3 execute in software on a dedicated computing facility located close to the detector [8]. The final output from the trigger is immediately stored and replicated across the world by the LHC Computing Grid.

IDScan is a track-reconstruction package running in Level 2 of the trigger, taking its input from the inner detector (shown in fig. 1.2), which has passed through Level 1. This project focusses on the z-finder algorithm, the first algorithm run in IDScan, described in §3.

1.2 Motivation

After the LHC has been operating for several years, it is intended to perform a major upgrade involving a ten-fold increase in luminosity; the number of particles passing through a unit area in unit time. This will be achieved through a combination of increasing the number of particles per bunch and decreasing the diameter of the beam at the points of collision.

This increased luminosity will lead to a similar increase in the number of events at each crossing. The intention is to allow for rarer, and possibly previously unseen, interactions to be observed and studied. The drawback of such a change is that the current trigger would not be capable of handling so high a throughput of events.

Many alternatives exist for increasing the capacity of the trigger after the upgrade. This project investigates the feasibility of accelerating Level 2 of the trigger using GPUs. The algorithm being investigated is one of the better suited for GPU. If potential for speed-up is shown, this opens up the possibility of using GPUs in a wider context in other detector software areas. Conversely, if speed-up appears unattainable, it would suggest that experimenters would be best investigating alternative methods, such as a simpler parallel paradigm such as OpenMP, or moving some of Level 2 onto a hardware implementation.

Whereas high performance computing usually concentrates on applications with long execution times, the trigger is used for online operations and is time critical, so requires high throughput and the lowest possible latency.

Chapter 2

GPGPU

General-purpose computing on graphics processing units (GPGPU) is a relatively recent development in HPC. Graphics processing is a computationally intensive, highly parallel operation. GPUs are highly specialised for this task, sacrificing memory caching and sophisticated flow control in favour of floating-point performance. Because of this, GPGPU is most suited to problems which can be performed in a data-parallel manner, in which similar operations are performed on many data elements. Fortunately, such problems can be found in many fields outside of image rendering, from fluid dynamics to finance.

There are several different APIs and SDKs which can be used to perform GPGPU. The first released was NVIDIA's CUDA [3] in November 2006. AMD/ATI offer a similar solution for their hardware with the Stream SDK. More recently, other frameworks have been developed which are capable of utilising hardware from multiple vendors; notably OpenCL, originally developed by Apple, and DirectCompute, part of Microsoft's DirectX framework.

CUDA was chosen for this project as it is the most popular and mature of the available solutions. Additionally, as it specific to NVIDIA hardware, it provides a fairly low-level interface to the hardware, giving potential for some more advanced optimisations. The CUDA architecture is described in detail in [6].

2.1 The CUDA architecture

There are now several generations of CUDA-capable graphics devices, with their different features specified by their so-called *compute capability*. This project will target the two most recent incarnations of the CUDA architecture, commonly referred to as the Tesla and Fermi architectures.

There is some potential for confusion here, as both cards are marketed under the Tesla brand, but with the newest cards having been referred to as Fermi during development.

To avoid this, I will refer only to devices with compute capability 1.3 as Tesla, and those with compute capability 2.0 as Fermi.

The Fermi hardware is the first major update to the CUDA architecture since its initial release. It introduces several features to improve programmability, such as a cache system for global memory (see below), as well as other features necessary for scientific applications, including good double precision floating-point performance and Error Correcting Code (ECC) memory.

Whilst Fermi presents the future of the CUDA architecture, the hardware was not available until late in the project. It was not possible to focus solely on this device, and the code is not as well optimised for Fermi as it might otherwise be.

2.1.1 Processing cores and multiprocessors

A diagram of the Fermi hardware can be seen in fig. 2.1. It can be seen in the image that there are hundreds of processing cores present (pictured in the darker green). The cores are grouped together in *multiprocessors*, which form the basis for the repeating pattern visible. Each multiprocessor has a selection of memory local to it, including registers and shared memory (see §2.1.4).

2.1.2 Kernels

A CUDA kernel is a function which is executed in parallel by a number of threads on the GPU device. CUDA is generally programmed for in C for CUDA; an extension to the C99 dialect of the C language, adding many type qualifiers, built-in variables and functions relating to CUDA, plus a few C++ features, such as templates.

The following kernel performs a simple vector addition in parallel on a single thread block:

```
1 __global__
2 void vector_add(float* A, float* B, float* C)
3 {
4   int i = threadIdx.x;
5   C[i] = A[i] + B[i];
6 }
```

The `__global__` declaration informs the compiler that this is a CUDA kernel. Each thread reads a single element from the `A` and `B` arrays, performs the addition, and writes the result to the array `C`.

The input data, arrays `A` and `B` must be explicitly transferred onto the GPU device using a `cudaMemcpy` call before the kernel can execute. Afterwards, the result must be retrieved in the same fashion.

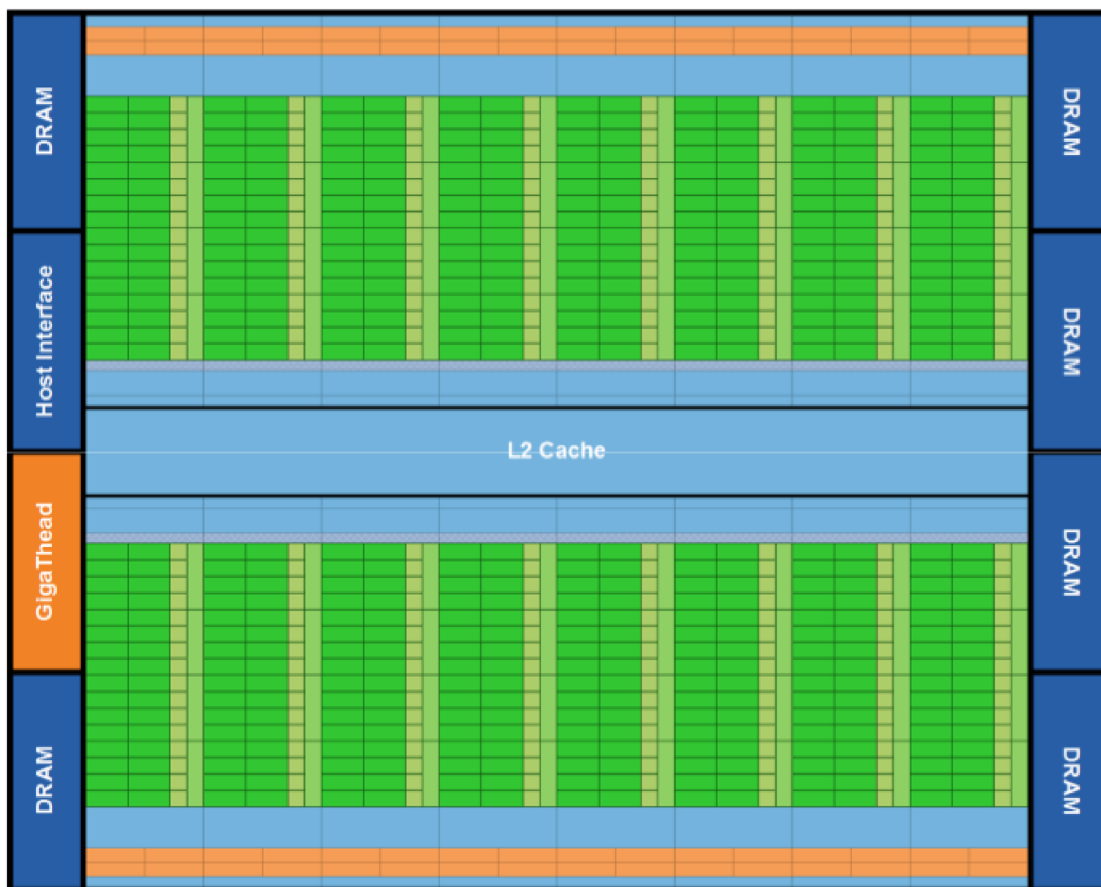

Figure 2.1: Diagram of Fermi hardware. Image from [7].

The kernel is invoked as a normal C function, but with the additional `<<<...>>>` syntax specifying the thread layout, for example `vector_add<<<1, N>>>(A, B, C);`

This invocation dictates that the kernel is to be run on a single block of `N` threads.

2.1.3 Thread hierarchy

CUDA threads are arranged in multiple levels, as seen in fig. 2.2, revealing the limitations of the device whilst attempting to give the programmer maximum flexibility.

Threads

As with CPUs, GPUs are capable of running parallel threads of execution, each with their own instruction address counter and register state. GPU threads are also able to branch and execute independently, though this may affect performance considerably (see **Warps** below).

Unlike CPUs however, GPU architectures are designed for running thousands of threads in parallel. This enables a program to fully utilise the hundreds of processing cores and high memory bandwidth of the GPU. As this multi-threading capability is built into the hardware, there is no additional overhead from running so many threads, and it is not unusual for a thread to calculate only a single element of an output array.

Each thread is given a unique identifier within its block, which may be accessed using the built-in `threadIdx` variable. For convenience, threads may be arranged in a virtual topology with up to three dimensions.

Thread blocks

A thread block is a set of threads which execute together on a single multiprocessor. The dimensions of the block topology may be accessed within a kernel using the inbuilt `blockDim` variable.

The threads within a thread block may be synchronised with a single lightweight function call, implemented in hardware. This, along with shared memory, allows for cooperation between threads in a single block, for instance with calculating or loading common values. No global thread barrier is present, however, as it cannot be guaranteed that all blocks will be running concurrently.

Multiple thread blocks may reside together on a single multiprocessor, provided there is sufficient shared memory and register space. Thread blocks are scheduled automatically by the hardware, allocating blocks to any multiprocessor which has the capacity to execute them.

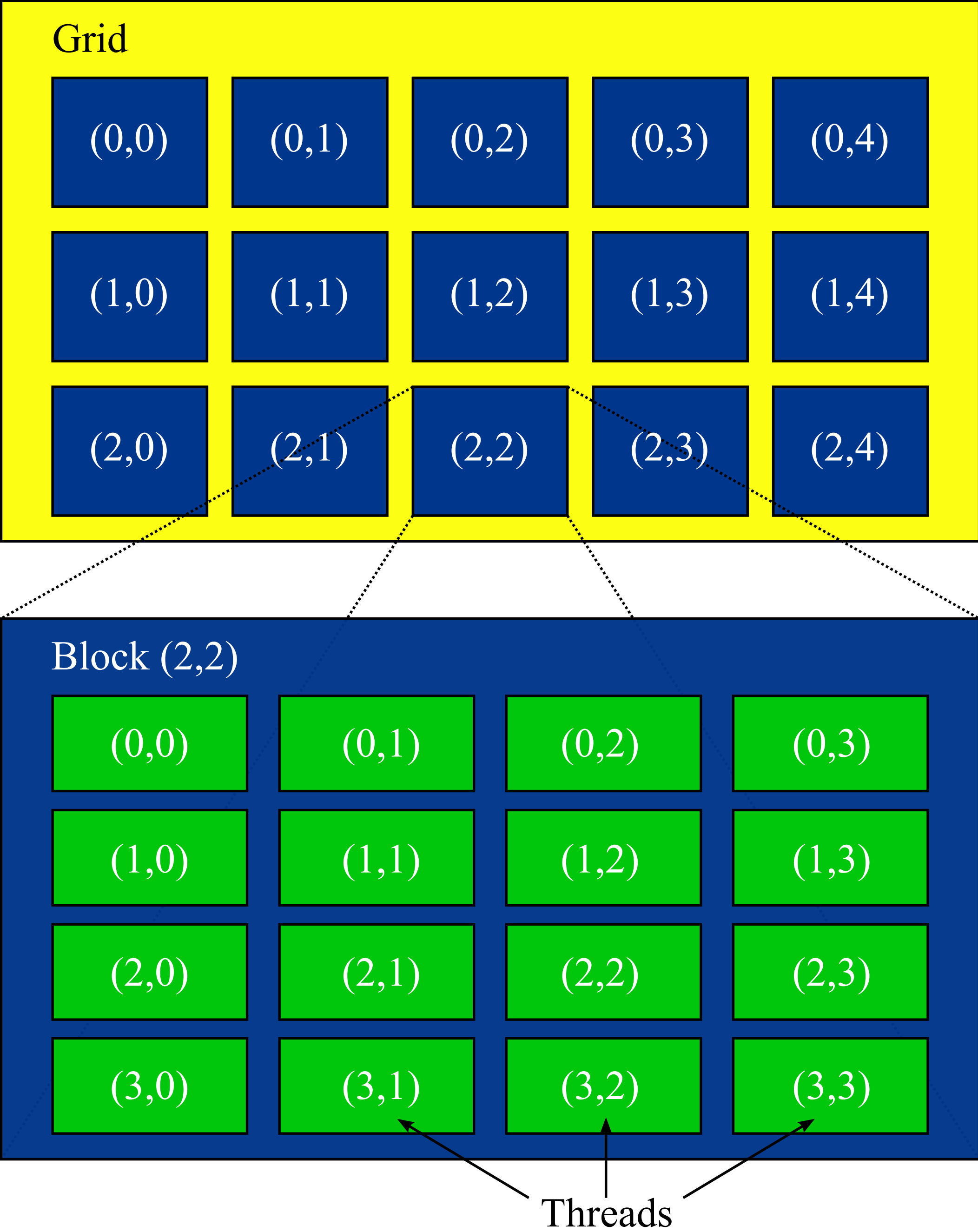

Figure 2.2: The CUDA thread hierarchy.

Blocks are arranged into a one- or two-dimensional grid, the dimensions of which may be accessed using the `gridDim` variable. The block's coordinate within that grid may be accessed via the `blockIdx` variable.

Typically, a thread might therefore calculate its coordinate in a 1D global thread space as `int gid = blockDim.x*blockIdx.x + threadIdx.x;`

Warps

The multiprocessor schedules and executes threads in sets of 32, called warps. A warp can only execute a single instruction at a time. In this way, a multiprocessor closely resembles a SIMD vector architecture. If the execution path of any threads within a warp diverge due to a conditional branch, the different paths are serialised until they again converge. For efficient performance, it is therefore important to avoid divergent warps wherever possible.

The warp scheduler automatically schedules warps which are ready to execute. Switching context between warps is done at no cost, as the registers for all running threads are maintained for the entire lifetime of their warp.

If a warp is stalled — reading from global memory, for instance — the warp scheduler is able to hide some of the latency by scheduling another warp; keeping the hardware busy. The warp scheduler can only do this effectively if the number of warps on the multiprocessor, the *occupancy*, is sufficiently high. An occupancy of 25% of the maximum is the recommended minimum, as this allows for register dependencies to be completely hidden [5].

On pre-Fermi architectures, the first half of the warp executes, immediately followed by the second half. Memory accesses also occur by the half-warp. On Fermi, all 32 threads execute concurrently.

2.1.4 Memory architecture

CUDA devices contain many different types of memory, each with their own properties. As so many computational codes are memory-bound, it is imperative to understand the performance characteristics of the most commonly used types.

Some of these memory types, such as registers and global memory have direct parallels in CPU architectures. Others, such texture memory, betray the graphical heritage of the hardware, and may seem quite unfamiliar.

Registers

Variables declared within a kernel are usually stored in registers; each thread having its own copy. There are thousands of registers on each multiprocessor, but these must be

divided up between all of the threads in a block.

As on a CPU, there is zero cost for a read from a register, except in cases where there is a data dependency from a previous instruction.

Global memory

The global, or device, memory is the main memory storage on the GPU, and comfortably the largest; several gigabytes on high-end cards. It cannot be accessed directly by the host. Instead, data must be explicitly copied between host and device using CUDA API functions.

Unlike a usual CPU memory architecture, pre-Fermi cards possess no caching system for global memory. As global memory reads typically have a latency of several hundred clock cycles, it is very important to avoid unnecessary global memory accesses.

Requests to global memory are issued for multiple consecutive words of memory simultaneously. For optimal performance, threads in a warp should request consecutive words of memory. Furthermore, the address read by the first thread should be aligned such that it is an exact multiple of the width of the request. A request which fulfils both these requirements is said to be coalesced.

Uncoalesced reads from memory will result in additional memory accesses, wasting memory bandwidth and impacting performance. The mechanism for the requests is quite complex, and varies substantially between different versions of the CUDA architecture [6].

Shared memory

Shared memory is a small area of memory[1] located on each multiprocessor. It enables efficient cooperation between threads in the same block, allowing them to share data between themselves.

Shared memory is arranged in memory banks, which can be accessed simultaneously, and shared memory addresses are distributed between them in a cyclic manner. Where two threads attempt to simultaneously read an address in the same bank, a bank conflict occurs and the request are serialised; the one exception being where all threads in a warp read the same address, in which case the value is broadcast.

Where no bank conflicts occur, the latency of a shared memory access is roughly 100x lower than that of an un-cached global memory access. In some circumstances, shared memory can provide an effective mechanism of manually caching the data used by many threads.

[1] 16KB on pre-Fermi architectures and up to 48KB on Fermi.

Local memory

Perhaps the most confusing memory type, local memory is so-named as it stores data from a thread's local scope, not because it is stored locally in hardware. In fact, access to local memory costs as much as a global memory read, and is to be avoided wherever possible. Like global memory, local memory is not cached on pre-Fermi architectures.

Local memory primarily acts as an overflow for variables which cannot fit into registers, such as large structs. It is particularly worth noting that any array declared in local scope will be placed in local memory.

Constant memory

Constant memory is a small area (64 KB) of read-only device memory with the advantage of being automatically cached. After the initial cost of a cache miss, a read from constant memory is as fast as a read from a register, provided that all threads of a half-warp read the same address. Accesses to different addresses within a half-warp will be serialised.

Texture memory

Texture memory is not so much a different class of memory as it is a different method of reading from global memory. Specific functions are used to retrieve data from a texture reference, which must first have been bound to an address in global memory. Like constant memory, texture memory is read-only and cached.

Texture memory is designed for streaming fetches with a constant latency, so a cache hit will not reduce fetch latency, but will reduce bandwidth demands on the device memory.

Texture reads can be very beneficial in accessing device memory in a non-optimally coalesced fashion, particularly on older GPU hardware with stricter coalescing requirements. In particular, texture reads are optimised for instances where the reads have an inherent 2D locality.

L1 and L2 cache

Present only on the new Fermi architecture, these caches act much like the equivalent caches in a typical CPU system. An L1 cache is present on each multiprocessor, whilst the L2 cache is shared between all multiprocessors. The cache works for reads from both global and local memory.

On Fermi, there is a total of 64 KB of on-chip memory to be divided between the L1 cache and shared memory. The developer chooses the division, but a maximum of 48 KB can be allocated to either.

Memory summary

A summary of the memory architectures can be seen in table 2.1.

	Tesla	Fermi
Number of 32-bit registers per multiprocessor	16 K	32 K
Maximum amount of L1 cache per multiprocessor	n/a	48 KB
Maximum amount of shared memory per multiprocessor	16 KB	48 KB
Number of shared memory banks	16	32
Local memory per thread	16 KB	512 KB
Constant memory size	64 KB	
Constant memory cache size per multiprocessor	8 KB	
Texture memory cache size per multiprocessor	Device dependent, 6–8 KB	

Table 2.1: Summary of memory architecture on the two GPU devices.

Chapter 3

Z-finder Algorithm

At high luminosities, many collisions occur at each bunch crossing, though there is likely to be no more than one high-p_T event[1], in which interesting physics occur. The location of the high-p$_T$ collision is called the primary vertex. An accurate estimate of this point is of interest as it can improve performance and reduce execution time of the track-reconstruction algorithms which run later in the Level 2 trigger. The algorithm is described in detail in [9].

The particle bunches are focussed into a very narrow beam in the collision area, so the coordinates of the primary vertex are known in two dimensions. The vertex will, however, vary within around 15 cm of the centre of the detector along the beam axis. It is the purpose of the z-finder algorithm to locate the z-coordinate of the primary vertex, z_0, to an accuracy of $\lesssim$1 mm.

Level 2 of the trigger operates only on data within regions of interest (RoIs) identified by Level 1. The coordinates of each detection in the ATLAS sensors are referred to as spacepoints. The z-finder algorithm operates on the spacepoints within a single RoI, returning a single vertex. Spacepoints use a cylindrical polar coordinate system, matching the natural geometry of the detector.

The inner detector has sensors located in 19 concentric, cylindrical layers. The algorithm works by calculating the z-coordinate of the intercept with the z-axis, z_V, of the trajectory through pairs of spacepoints in different layers of the detector. The simplest method is a linear extrapolation, which provides a good approximation:

$$z_V = \frac{z_2 \cdot \rho_1 - z_1 \cdot \rho_2}{\rho_1 - \rho_2}$$

where ($z_{\{1,2\}}$, $\rho_{\{1,2\}}$, $\phi_{\{1,2\}}$) are the coordinates of the two spacepoints.

This coordinate is then inserted into a histogram known as the z-histogram. Most arbitrary pairings of two random spacepoints will not represent a genuine particle trajectory through the detector, so will add noise the resulting histogram. But those pairings that

[1] p_T is the transverse momentum.

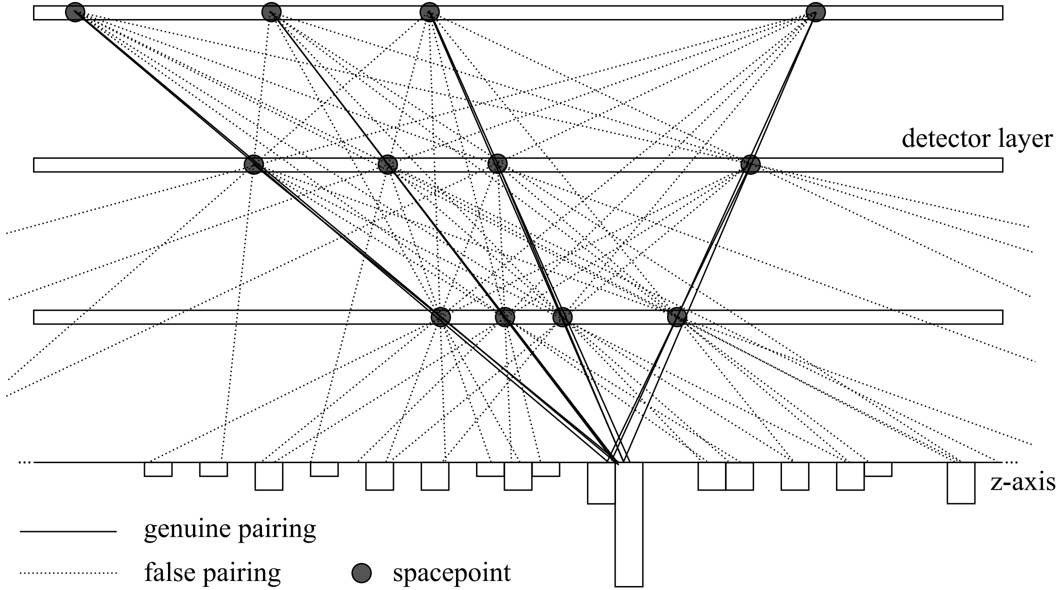

Figure 3.1: Schematic showing how z-vertices are calculated using pairs of particles within a ϕ slice.

are on a real track originating at the primary vertex will cluster around z_0, as can be seen in fig. 3.1. A real z-histogram for a low luminosity sample can be seen in fig. 3.2.

If all pairs of spacepoints in an RoI were considered, the peak around z_0 may get lost by combinational noise. Additionally, at high luminosity it is quite possible that multiple interactions will occur along the z-axis. But the only vertex of interest at this stage is the primary vertex, which will give rise to most of the high-p_T tracks in the RoI.

By dividing the spacepoints into thin slices in the ϕ direction we are able to solve both problems to some extent. We only use pairs within a slice, or one of its immediate neighbours, to calculate the z-vertex. High-p_T tracks will bend little in the magnetic field of the detector, so will give spacepoints with almost the same ϕ-coordinate. Conversely, tracks with lower p_T will not remain within thin ϕ slice, so will not contribute the final histogram. The slice size used is around $0.2°$, with a typical RoI spanning around $100°$.

3.1 Using triplets of spacepoints

Even having split the spacepoints into ϕ slices, at high luminosity, peaks can be lost amongst the combinational noise, such as in fig. 3.3. This is because while the number of real spacepoints tracks will be $O(N_{sp})$, the number of total pairings will be $O(N_{sp}^2)$, where N_{sp} is the number of spacepoints.

A solution to this problem is to consider triplets of spacepoints instead of pairs. After

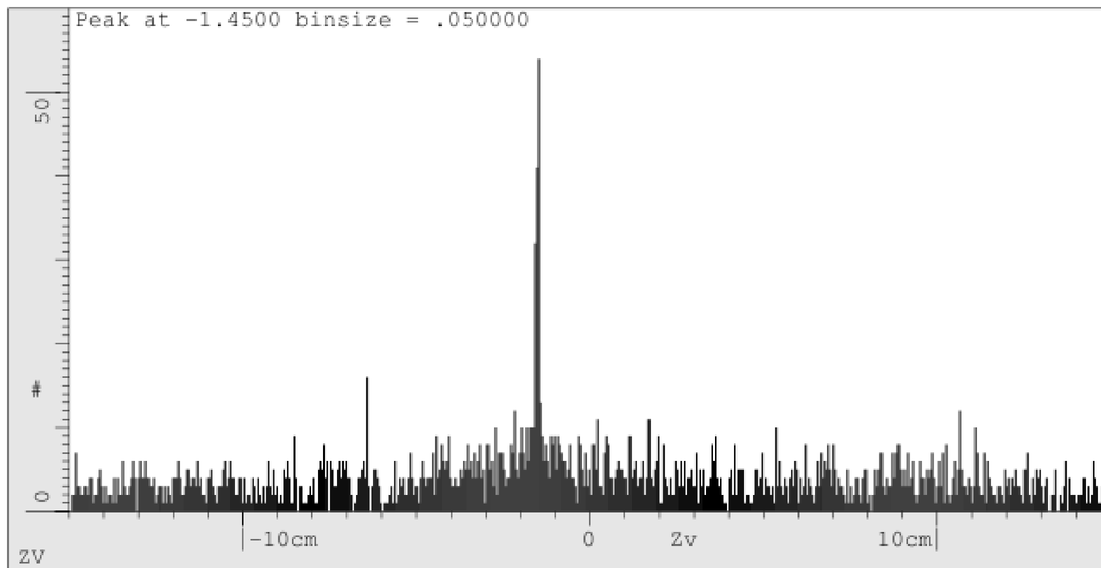

Figure 3.2: The z-histogram for a low luminosity sample. Image from [9].

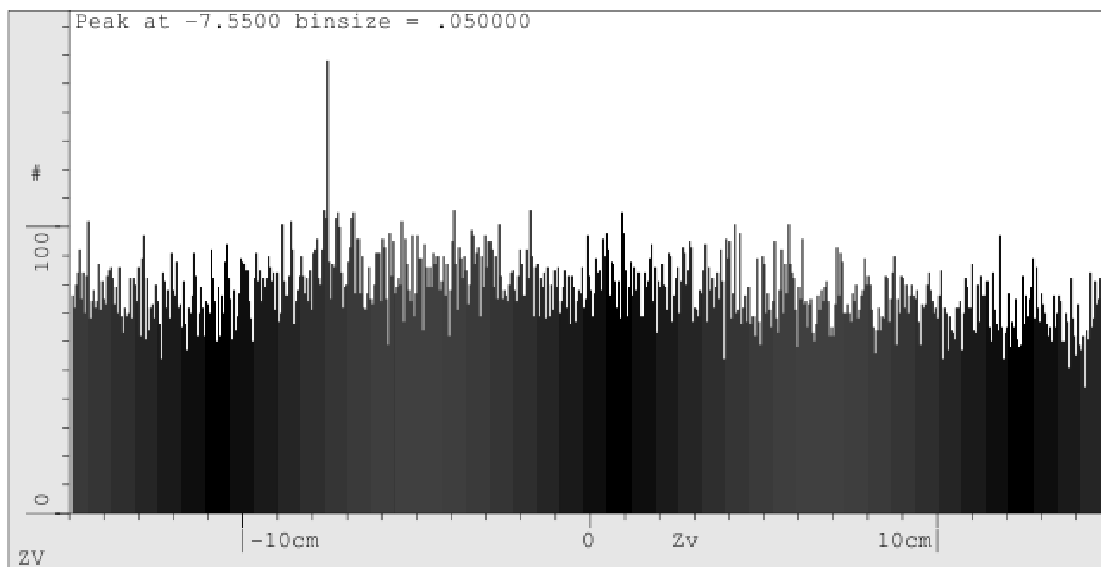

Figure 3.3: The z-histogram for a high luminosity sample. Image from [9].

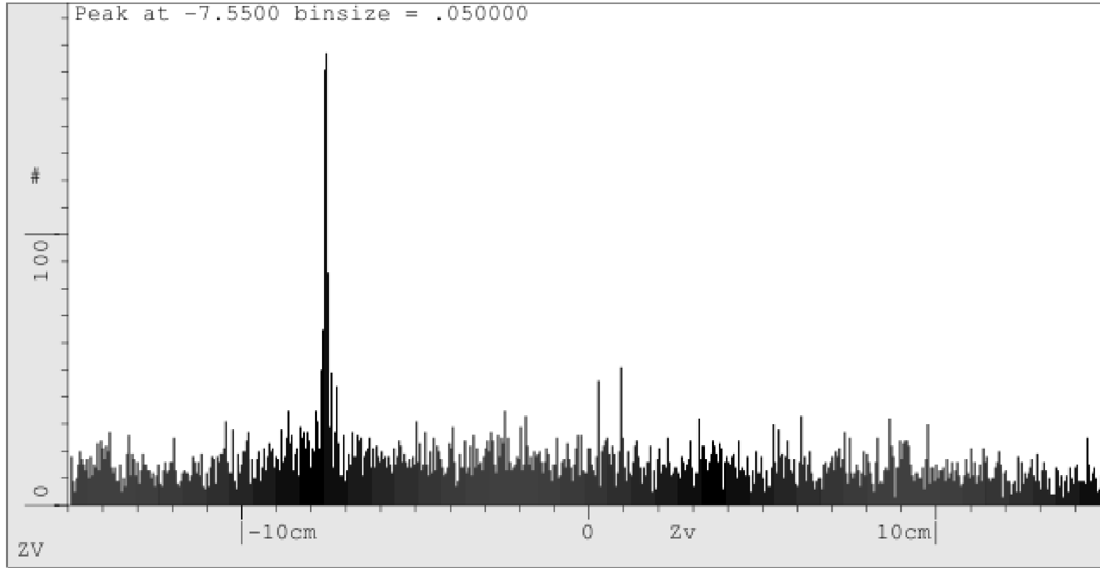

Figure 3.4: The z-histogram for a high luminosity sample, using triplets of spacepoints. Image from [9].

a z_V value has been calculated for a pair of spacepoints, the points in higher layers are checked for potentially being on the same track, within a predetermined tolerance. If no triplet is found, then the vertex is not added to the z-histogram. A z-histogram constructed only from triplets can be seen in fig. 3.4.

3.2 Serial code implementation

As mentioned above, the algorithm requires the spacepoints to be divided into ϕ slices. Ideally, the spacepoints also need to be grouped by detector layer. This could be achieved using two sorts of the spacepoints, but these are fairly expensive operations; typically $O(N \log N)$.

Instead, for each layer in every slice, a list is constructed of all the spacepoints present within. This allows the 'sort' to be performed in a single pass; i.e. $O(N)$. An array, `inext` is created, whereby if spacepoint `n` is in a list, then `inext[n]` is the index of the next spacepoint in the same list. Arrays are also constructed to store the first spacepoint in each list, and the number of spacepoints in each. This implementation is in many ways similar to a linked list, but uses a fixed, predeterminable amount of memory.

Each spacepoint is read in turn and inserted at the beginning of the appropriate list. It is prependend, rather than appended, as this does not require traversing the list, so is much more efficient. If spacepoint `n` is added to a list, `n` becomes the first index in the list and `inext[n]` is set to the previous first index. The counter for the list is then incremented.

In practice, z_0 is actually calculated from two 'histograms', called `nHisto` and `zHisto`.

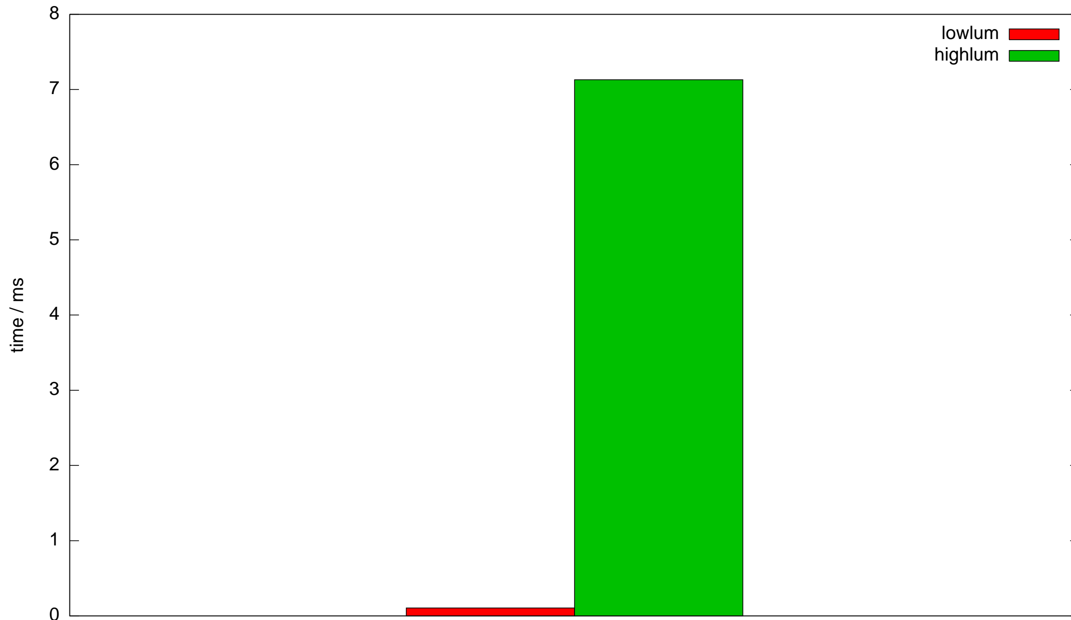

Figure 3.5: Execution time for serial code.

The first is a conventional histogram storing the vertex count for each bin. The second stores the floating-point sum of all the vertices in each bin. By preserving the data in the latter, we are able to calculate a more accurate value for z_0 without having to substantially increase the number of bins in the histogram.

To calculate the peak, the three adjacent bins in the `nHisto` with the largest total count are located. z_0 is then the sum of the three corresponding bins in the `zHisto` divided by this count. If a second vertex is requested, these three bins are simply set to zero, and the search is repeated.

The performance of the serial version of the code may be seen in fig. 3.5.

Chapter 4

Implementation and Benchmarking

4.1 Implementation

The z-finder algorithm is normally contained as part of a much larger framework, and as such it was non-trivial to isolate the performance results needed. To facilitate this, a wrapper program was written to allow the z-finder algorithm to be executed in an isolated manner, and output exactly the information needed.

The internals of the z-finder algorithm are contained within the `IDScanZFinderInternal` class in a header file. The GPU version of the code was formed in a separate copy of this header file, and the appropriate version included depending on which compiler was used to compile the code. This allows for seamlessly switching between the CPU and GPU implementations.

The serial implementation of the code contains many configurable properties at runtime, but it was decided to concentrate on only a few. Many simplifications were made to the code in porting to GPU; focussing on a purer implementation of the z-finder algorithm as described in §3. These included limiting the output to a single global array and only considering single neighbouring slices for the calculations.

To ensure that the results of the two implementations are directly comparable, the serial code was brought in line with the simplified GPU code. In most cases, the additional serial code functionality could be switched off simply by overriding member variables of the `IDScanZFinderInternal` class. This was performed by producing a subclass, as is the intention, and overriding these variables there. This way, the serial code calculation was made to match that of the GPU code with very little alteration of the original.

A very preliminary attempt at porting the z-finder code to GPU had been made by researchers at the Rutherford Appleton Laboratory, though no GPU code itself had been written. The work utilised a `slice` struct, containing the information about the space-points in the slice and the histograms for the slice. The same approach was also adopted in this port, though the histograms were separated out to prevent the unnecessary trans-

fer of large empty arrays to the GPU. Initially, the `slice` struct maintained pointers to the histograms instead, but ultimately these were dropped also.

I wished to be able to switch seamlessly between single and double precision floating-point arithmetic within the GPU code. The facilitate this, the **`real`** type was defined, which is equivalent to a **`float`** or **`double`** depending upon a pre-processor flag given at compile time. **`uint`** was used frequently as a shorthand for **`unsigned int`**.

In the host code, pointers to arrays on the GPU are prefixed with `d_`. Similarly, in GPU code, shared memory arrays are prefixed with `s_`. All other variables can be assumed to reside in registers or main memory. The `IDScanZFinderInternal` class uses the `m_` prefix to label member variables of the class.

Initially, only the part of the code which constructs the z-histogram was ported to GPU. This will be referred to as the *findZ* kernel. Later, the histogram summation was also moved onto the GPU; this kernel being called *sumHistos*. The other code was left unchanged, wherever possible.

4.2 Benchmarking

Timing values for each run were calculated by taking the time for each section of code separately, which were then averaged over all of the RoIs in the input file. The final timing result was then taken as the mean of three runs. Kernels were timed using the CUDA event timer, which provides greater precision and consistency than the CPU timer. The CUDA event timer is used as follows:

```
1   cudaEventRecord(start, 0);  // set start event
2   kernel<<<...,...>>>(...);   // launch kernel
3   cudaEventRecord(stop, 0);   // set stop event
4   cudaEventSynchronize(stop); // wait for stop to occur
5   cudaEventElapsedTime(&deltat, start, stop);
```

The second call to `cudaEventRecord` places the stop event immediately after the conclusion of the kernel, but both the kernel and `cudaEventRecord` return immediately. We must use `cudaEventSynchronize` to wait for the event to have actually occurred before calculating the elapsed time.

The CPU timer was used for all other measurements. When timing the entire algorithm, all other timing measurements were switched off using a preprocessor switch, eliminating the cost of the timing functions and printed output.

4.2.1 Machines

Performance measurements were performed on two different machines; one containing a Tesla card, the other a Fermi. Both machines contain Intel Xeon processors of reasonably similar performance, as seen in table 4.1. The CPU on the Tesla machine, being the marginally faster of the two, was used for timing the serial code.

	Tesla	**Fermi**
CPU Model	Intel Xeon X5560	Intel Xeon X5650
CPU Frequency	2.80 GHz	2.67 GHz
GPU Model	NVIDIA Tesla C1060	NVIDIA Tesla C2050
Number of multiprocessors	30	14
Number of processing cores	240	448
Frequency of processing cores	1.3 GHz	1.15 GHz
Single-precision peak performance	933 GFlops	1.030 TFlops
Double-precision peak performance	78 GFlops	515 GFlops
Global memory speed	800 MHz	1.5 GHz
Global memory bandwidth	102 GB/s	144 GB/s

Table 4.1: Specifications of the two benchmarking machines. Data from CUDA SDK's deviceQuery program and [3].

4.2.2 Input data

Two different simulated data samples were used for benchmarking the code: a lower luminosity sample, representative of the kind of data being produced by ATLAS in 2010, and a higher luminosity sample, representative of the data which might be received by the algorithm after the proposed hardware upgrade.

The simulated data sets are divided into different sets of physical processes. Both samples used were designated as *Bphysics* samples; those involving interactions of the B meson. This was largely chosen due to the large number of RoIs of this type generated by the simulation. The exact physics represented should be of little consequence to this algorithm.

	lowlum	**highlum**
Number of RoIs	650	1177
Mean number of spacepoints	333	8104
Mean number of ϕ slices	517	542

Table 4.2: Properties of the two input data files, with mean averaged over the RoIs.

Chapter 5

FindZ Kernel

The findZ kernel is the core of the z-finder algorithm, so it is clearly fundamental that we attempt to extract maximum performance from it.

5.1 Floating-point precision

The serial version of the code performs all of its calculations in double precision. On a CPU, double precision performance is typically half that of single precision and the same is true on Fermi devices. However, as we can see in table 4.1, on Tesla devices, double precision performance is significantly lower than single precision, and may prove unnecessarily costly. Additionally, much of the data to be transferred between host and device is in floating-point format. By converting to single precision prior to the transfer, the volume of this data is halved.

Before this conversion can be made, we must ensure that the results generated using single precision are sufficiently accurate. The vertex given by the z-finder algorithm need only be accurate to around 1 mm, as the value will be refined by the track reconstruction algorithms at a later stage. As the possible range for the vertex is a couple of hundred millimetres, this corresponds to only three or four significant figures of accuracy necessary. Single precision, providing six or seven places of accuracy, should therefore be sufficient. The output of the detector itself will not be accurate to double precision in any case.

The errors of the serial code and the GPU code running in single and double precision can be seen in table 5.1. As the true vertex is not known, this was calculated with the serial code using its more accurate calculation for z_V; i.e. not using the linear approximation given in §3. A low luminosity sample was used to make the comparison, as this excludes the possibility of the correct vertex being lost in combinational noise, so each version of the code will at least be returning the correct vertex.

We can see from the table that the results of the double precision GPU code do not

Code version	Error
CPU, double precision	0.031 ± 0.091 mm
GPU, double precision	0.116 ± 0.116 mm
GPU, single precision	0.120 ± 0.120 mm

Table 5.1: The error in the vertex when using different versions of the code.

exactly match those of the CPU code. This is because the CPU code actually uses one more bin than the GPU code, and they are offset by half a bin width. This doesn't necessarily mean that the GPU code is any less accurate, but the error when compared to our reference is increased.

More importantly, we can observe that the errors in the two GPU versions are very similar. We can therefore justify performing the calculations in single precision. All timing results given for GPUS are for single precision.

5.2 Parallelism between thread blocks

As mentioned previously in §3, each ϕ slice is operated on independently. This is extremely beneficial when porting to the GPU, as we can allocate a thread block to each slice, with no need to communicate between them.

In the first attempt for this kernel, the original z-vertex calculation from serial code was ported to GPU, but with many simplifications. Only a single thread was able to execute this code in each block, as several additions were necessary to allow the threads to execute parallel calculations, as seen in the following section. Multiple blocks can however run simultaneously on different multiprocessors, so there is a performance benefit even with this very simple approach, as now the calculations for different slices are being performed in parallel.

5.3 Parallelism between threads in each block

To get close to exploiting the potential computing power of the GPU, we must enable the code to run on multiple threads within each block. There are two obvious strategies for doing so: either allocate a slice to each thread, or continue allocating an entire block to each slice and find a way to divide the work between the threads.

The first of these strategies will not be at all efficient for several reasons:

- there would be insufficient threads blocks to effectively utilise all of the multiprocessors.
- the work load between different slices is extremely unbalanced, so this approach

would result in highly divergent warps and a considerable number of inactive cores.

- there would be insufficient shared memory to copy data for all of the slices into shared memory, so these would need to be continually re-read from global memory.

This approach was therefore not pursued. Instead, it was necessary to find a method of splitting the calculations within a slice between the threads in each block. Though the calculations between pairs of spacepoints are independent, there are two main obstacles to achieving this parallelisation.

Firstly, the `zHisto`, the floating-point histogram mentioned in §3.2 must be constructed without floating-point atomic instructions, as these are not available on Tesla devices. Secondly, each thread must be able to calculate which two spacepoints it is using to calculate its z-vertex. This is made awkward by the manner in which the spacepoints are sorted into a 'linked list'. We consider these two problems in turn.

5.3.1 Constructing histograms in parallel

When constructing histograms in parallel, it is apparent that there is the possibility of two or more threads attempting to add to the same bin concurrently; particularly in bins around the peak. In CUDA, this would result in one or more of these writes being dropped and incorrect results being achieved. Where multiple threads attempt to write to a single address, only one of these writes is successful and all others are discarded.

In other parallel environments, such as OpenMP, such a problem might be circumvented through the use of atomic instructions, which in many modern architectures are implemented in hardware. On Tesla, atomic instructions are only implemented for integer variables, not floating-point variables. As described in §3.2, two separate histograms are contructed. `nHisto` is a standard integer histogram, whilst `zHisto` is a floating-point histogram. Therefore, whilst `nHisto` can be constructed using atomics, there is no simple method for constructing `zHisto`. Suitable atomic instructions are present on the Fermi architecture, at least for single-precision, so the histogram construction presents no problems there.

The simplest solution to this problem would be for each thread in a block to have its own `zHisto`, turning `zHisto` into a two-dimensional array in shared memory, then performing a reduction over this array at the end. However, the limited quantity of shared memory present would necessitate reducing the number of bins in the histogram to an unacceptably low level.

A more practicable solution is to attempt to create a mutex lock to protect access to `zHisto`. Creating a single lock though would be extremely expensive; serialising all writes to the histogram, plus the additional overheads. Instead, we create a separate mutex for each bin in the histogram. Whilst this does require us to use more shared

memory, it allows the majority of writes to the histogram to occur in parallel, serialising only where multiple threads must write to a single bin.

One *incorrect* method of implementing such a mutex lock, might employ a spinning **while** loop as follows:

```
1 while (1) {
2   if (!atomicExch(&(s_locks[nz]), 1)) {
3     myZHisto[nz] += zv; // add vertex to histogram
4     __threadfence();    // wait for write to complete
5     s_locks[nz] = 0;    // unset lock
6     break;
7   }
8 }
```

Each thread is attempting to add its vertex, `zv`, to the bin with index `nz`. `s_locks` is an integer array containing our locks; `1` meaning set, and `0` meaning unset.

The `atomicExch` function overwrites the value pointed to by the first argument with the second argument, returning the value previously contained. The **if** statement therefore will only evaluate as true if the lock was previously unset, i.e. `0`.

As previously implied, this code does not execute as expected on a GPU, due to the manner in which threads are scheduled. It is possible for one thread to get stuck in an infinite loop, continually waiting for another thread to release the lock, but with this thread not executing the code to do so.

In order to alleviate this problem, we must employ a different strategy, such as this:

```
1 bool added = 0;
2 do {
3   s_locks[nz] = tid;    // try to acquire lock
4   if (s_locks[nz] == tid) {
5     myZHisto[nz] += zv; // added vertex to histogram
6     added = 1;          // done
7   }
8 } while (!added);
```

In this method, each thread attempts to write a unique identifier, the thread's ID, to the lock. If multiple threads attempt to set the same lock, only one will be successful. The thread then checks if its write completed, and if so is considered to have acquired the lock. The `s_locks` array must be declared as **volatile** to ensure that the compiler does not optimise out the read in the **if** statement condition.

We can see there is no possibility of an infinite loop, as the lock does not require unlocking. This does present the possibility of there being incorrect behaviour if the scheduler executes the instructions in an unusual order, though in practice there is no evidence to suggest this, and the kernel consistently produces the same result as the atomic instruction on Fermi.

There is however yet another approach, possible only because we are considering floating-point values. The probability of two threads not only wishing to write to the same bin concurrently, but having calculated almost exactly the same the value for zv is negligible. Therefore, we can attempt to write directly into myZHisto, then check if the result is as expected. In doing so, we can discard the notion of mutex locks entirely.

```
real new_val;
do {
  new_val = myZHisto[nz] + zv;
  myZHisto[nz] = new_val;
} while (myZHisto[nz] != new_val);
```

As each thread will have a different value for zv, it will correspondingly calculate a different new_val. The latter assumption might become problematic were myZHisto[nz] to be many orders of magnitude larger than zv[1], but this will not occur as there will never be sufficient tracks pointing to the same bin within the same slice; with the high-lum sample the maximum bin count is around 100.

In this instance, myZHisto must be declared **volatile**, as we must force a re-read of this value. The robustness of this method is asserted by comparison of results using this method with the results using only atomic functions on Fermi.

5.3.2 Calculating pairs of spacepoints

The calculation of the vertex for each pair is independent, so should be trivially parallelisable. The difficulty comes from calculating which two spacepoints each thread should use for its calculation.

The main obstacle to doing this is the manner in which the data is sorted into slices (see §3.2). The only method of calculating the nth spacepoint in a layer is the calculating the $n-1$ spacepoints before it in the layer. This is a very costly operation, as calculating each point in the sequence will require a read from global memory. These reads cannot be coalesced and we cannot copy the inext array into shared memory as it may be too large.

Each spacepoint in the slice, and adjacent slices, will be needed multiple times by multiple threads. We can eliminate the need for repeatedly performing this procedure by producing a flattened version of the spacepoint indices in a single array. The array produced groups the points together by layer, with points from the central slice coming first.

The array we create in device memory, d_index, allows the two spacepoints in the pair to be read directly by each thread. The second spacepoint in each pair originating at a given spacepoint is calculated by starting at the first point in the layer above, then

[1]Due to the loss in precision when adding floating-point variables of different magnitude.

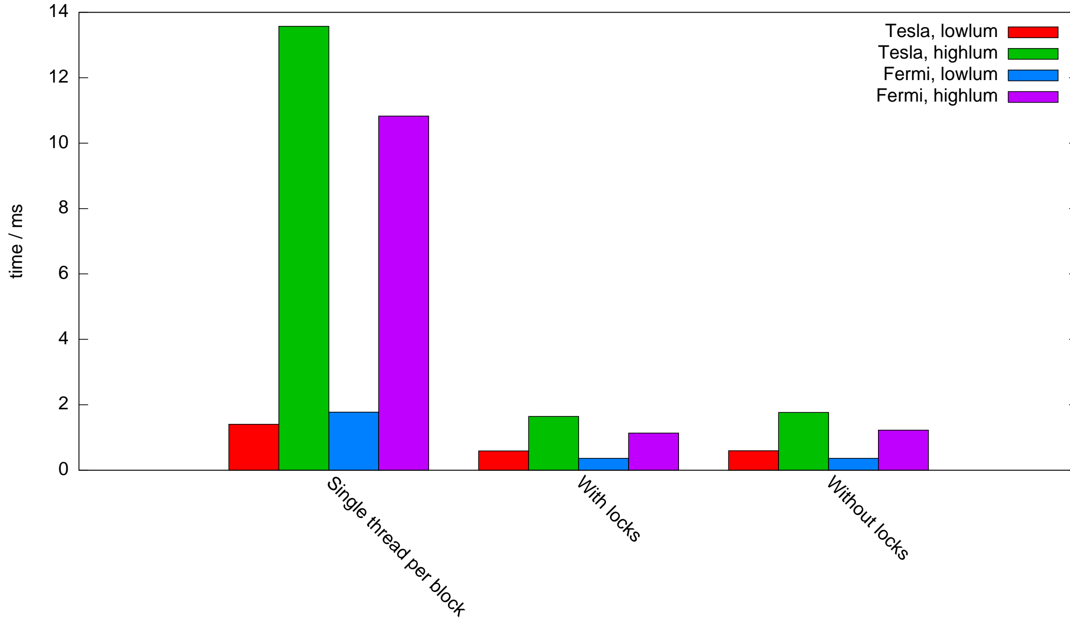

Figure 5.1: Execution time of single thread GPU code and initial thread parallel kernels.

looping over all the points that follow. This loop can trivially be split among the threads in a block.

An array, `s_ifirst`, is created containing the indices within `d_index` of the first point in each layer. This is done by performing an exclusive scan over the number of spacepoints in each layer within the slice. An additional element is present at the end of the `s_ifirst` array, which stores the total number of spacepoints in the three slice.

5.4 Kernels

The first thread parallel kernel loops over each layer, and over each point within that layer. The calculation of pairs incorporating this point is split amongst the threads in the block, as discussed above. This kernel was timed using both of the *working* histogram construction methods discussed in §5.3.1, and the results shown in fig. 5.1.

We see an immediate speed-up of around 3x for low luminosity and 8x for high luminosity with the move to a thread parallel kernel. The version using locks is marginally faster at this stage, as without locks, two reads of `myZHisto` are made, which resides in global memory. A comparison with the figures for the entire calculation in the serial in fig. 3.5, will give a good impression of how much further improvement is required.

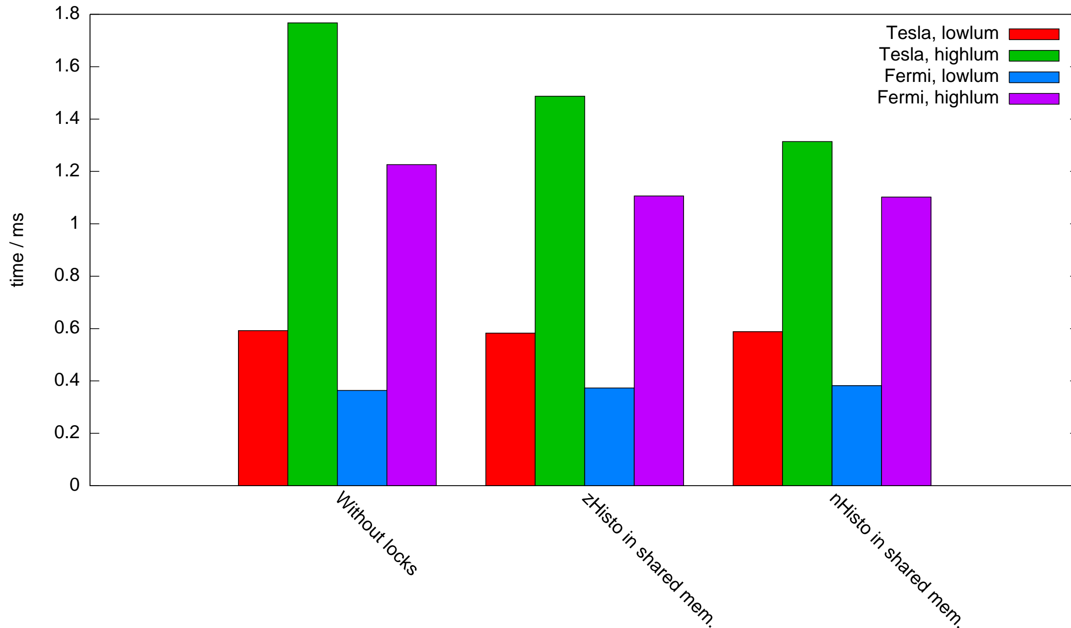

Figure 5.2: Execution time of findZ kernel with histograms in shared memory.

5.4.1 Histogram construction in shared memory

Re-reading values of histogram bins from global memory is an expensive operation. By moving the histogram construction into shared memory, we remove much of this cost. This is particularly true at high luminosity, as there are far more vertices to be added to the histograms. The benefit of moving `myZHisto` into shared memory is clear, in fig. 5.2, for both the Tesla and Fermi with high luminosity input. Moving `myNHisto` into shared memory appears of significant benefit only on the Tesla.

5.4.2 Optimising initialisation code

If we comment out the code calculating vertices, leaving only the initialisation code, we can see that on Tesla this code takes more than a third of the time for the kernel, even at high luminosity. On Fermi, the code is not so costly, as the caching system greatly benefits the unravelling of the 'linked lists' of spacepoints.

In the initial version of the kernel, the entire initialisation code is executed by only a single thread. As there are only 19 layers in the detector, there is limited potential for parallelising this part of the code, but most of it can utilise at least one thread per layer.

The results from various optimisations to the initialisation code can be seen in fig. 5.3.

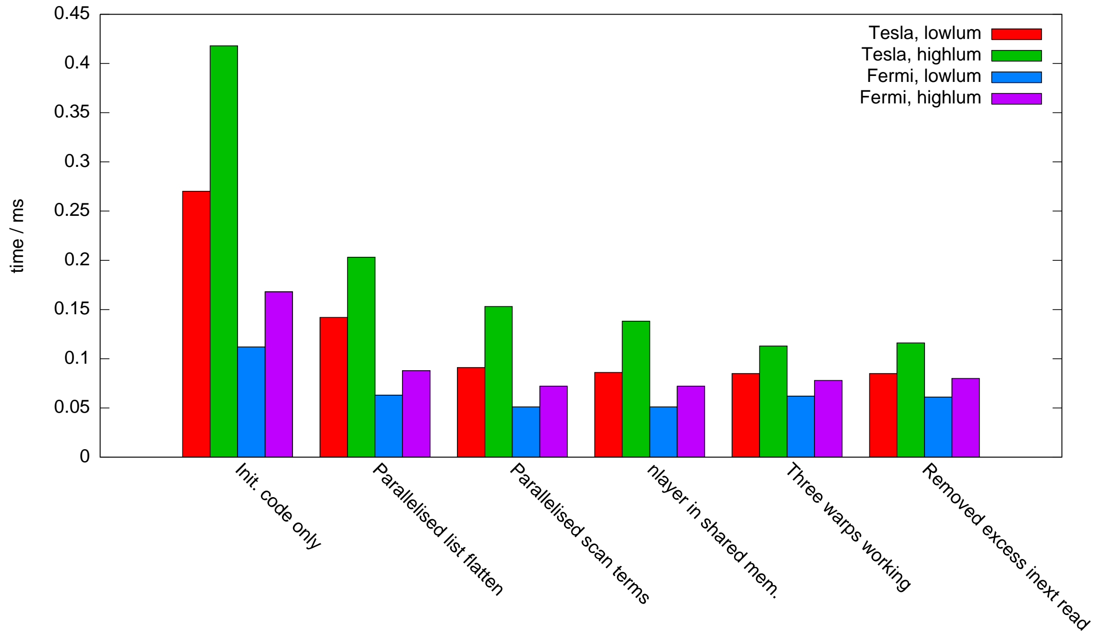

Figure 5.3: Execution times of the initialisation section of the findZ kernel.

Parallelising flattened spacepoint array construction

This part of the code is trivially parallelisable. Each of the first 19 threads copies the spacepoints from the three neighbouring slices into the appropriate point of the `d_index`. The initial offset can be read directly from the `s_ifirst` array, which is calculated first.

This optimisation approximately doubles the performance of the entire initialisation section of the kernel on both architectures.

Parallelising scan calculation

Another possible area for improvement is the scan used to create `s_ifirst`. Unfortunately though, whilst efficient parallel scan algorithms do exist, it is not worthwhile to use one on an array of only 20 elements. Even a manually unrolled version of the parallel algorithm would require just as many instructions as a serial version running on a single thread, and would complicate the code needlessly.

The scan can still be improved upon by ensuring that the single thread performing the scan only does the bare minimum of the calculation. By calculating each element of the scan in parallel prior the commencing the scan, we can see a noticeable improvement.

Moving nlayer into shared memory

The `nlayer` values for each of the three slices are all referenced more than once in the kernel. We can remove the need for a repeat read from global memory, by storing these values in shared memory.[2] This does result in a minor improvement in performance.

Three warps performing flattened spacepoint array construction

We can actually take the parallelisation of the construction of `d_index` one stage further. As the spacepoints must be read in from three slices, we can parallelise among 57 threads rather than just 19. If we did this by simple thread ID, we would suffer from divergent warps when calculating the initial offset.

Instead, we use the first 19 threads from each of the first three warps. This leads to a reasonable improvement on Tesla. Despite a slight degradation on Fermi, this optimisation was maintained.

Removing excess inext read

The last attempted optimisation of the initialisation code was to remove the surplus read of the `inext` array performed on the last iteration each time a point is added to `d_index`.

```
1      ...
2      uint hit = wslice->ifirst[wid];
3      if (s_nlayer[wno][wid] > 0) {
4        int i;
5        for (i = 0; i < s_nlayer[wno][wid]-1; i++, hit = inext[hit]) {
6          d_index[ifirst+i] = hit;
7        }
8        d_index[ifirst+i] = hit;
9      }
10     ...
```

Here, `wid = tid % 32` and `wno = tid / 32`, so represent the thread ID within the warp and the ID of the warp respectively.[3]

The last iteration of the loop is removed and performed separately. Surprisingly, this 'optimisation' results in a performance drop in all instances, so was removed before returning to consider the rest of the kernel.

---

[2] At this stage, the `nlayer` values for the adjacent slices could be stored in registers, as they are only referenced by a single thread. This is not the case in subsequent kernels, however.

[3] Typically, integer modulus and divide are expensive calculations on a GPU, but as 32 is a power of 2, in this case they compile directly to single bitwise operation.

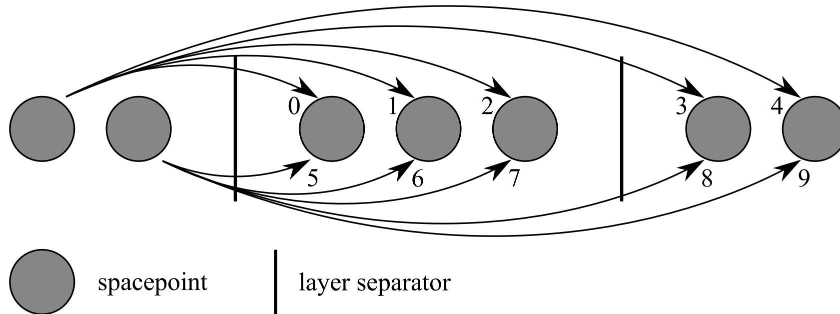

Figure 5.4: Schematic showing how each pair of spacepoints is numbered.

5.4.3 Calculating more pairs in parallel

Performing the vertex calculation as currently, where each thread uses the same lower-level spacepoint, is not likely to provide sufficient pairs to occupy all of the threads. The threads would be better utilised if all of the pairs originating at a layer, rather than a single spacepoint, could be calculated simultaneously.

To achieve this, the pairs in each layer are labelled sequentially from zero as shown in fig. 5.4. We then loop over each layer and loop over all the pairs originating from each layer.

If there are `n` points in the current layer, and `nAbove` points in all the layers above, there are a total of `n * nAbove` pairs to calculate in the layer. The thread can calculate which two spacepoints it needs using:

```
uint hit1 = d_index[s_ifirst[L1]   + pair / nAbove];
uint hit2 = d_index[s_ifirst[L1+1] + pair % nAbove];
```

where `pair` identifies which pair we wish to calculate and `L1` is the current layer. `hit1` and `hit2` are the indexes of the two spacepoints in the coordinate arrays.

The benefits of occupying more threads can be seen in fig. 5.5, particularly for the high luminosity samples.

Integer division and modulus are very costly operations on CUDA, as they are not implemented in hardware [6]. These operations can be removed at the expense of introducing divergent warps, resulting in a slight improvement overall.

```
uint index1 = s_ifirst[L1];
uint index2 = s_ifirst[L1+1] + pair;
while (index2 >= s_ifirst[IdScan_MaxNumLayers]) {
  index1 ++;
  index2 -= nAbove;
}
uint hit1 = d_index[index1];
uint hit2 = d_index[index2];
```

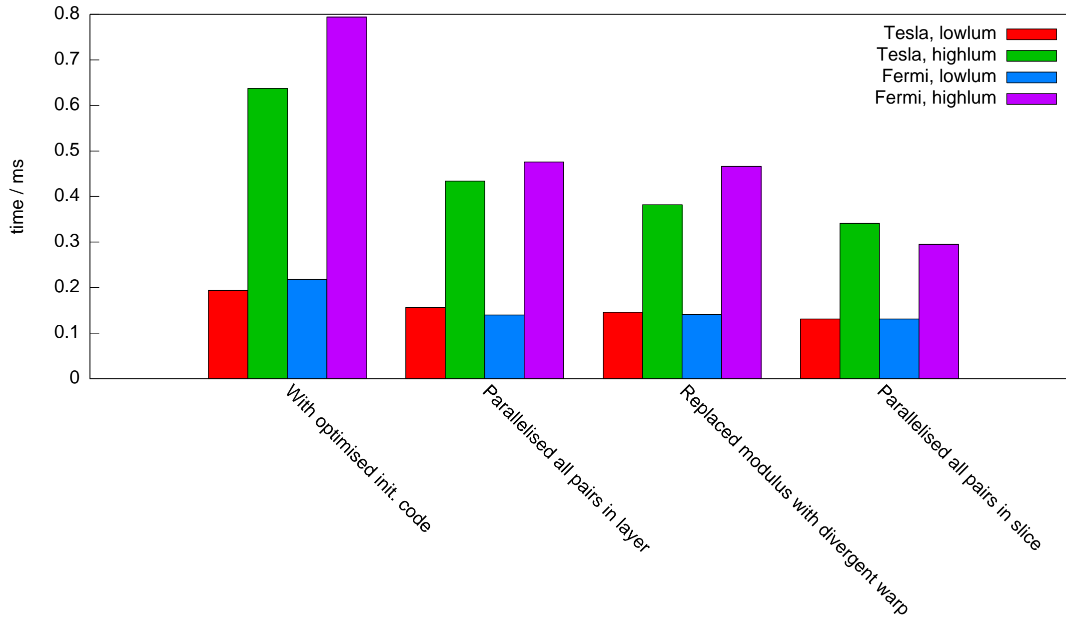

Figure 5.5: Execution times of the findZ kernel performing more work in parallel.

`s_ifirst[IdScan_MaxNumLayers]` contains the total number of spacepoints in the slice.

5.4.4 Parallelising all pairs in slice

One problem with looping over layers is that at higher layers there are fewer pairs to calculate with. This will inevitably leave most threads inactive in the upper layers. A potential solution to this problem is to label all of the pairs in the slice together, and avoid looping over the layers at all. We iterate over all of the combinational pairs in the slice with a single loop. By doing so, we add some additional overhead, but ensure the maximum possible utilisation of the available threads.

First, an array called `s_pfirst` was created, containing the number of the first pair in each layer. This is another exclusive scan, this time over the number of pairs in each layer. From this, a thread may calculate its layer, and then proceed with the calculation as before.

```
1   ...
2   uint L1 = 0;
3   for (uint pair = tid; pair < s_pfirst[IdScan_MaxNumLayers];
4         pair += blockDim.x) {
5     while (pair >= s_pfirst[L1+1]) L1++;
6
7     uint nAbove = s_ifirst[IdScan_MaxNumLayers] - s_ifirst[L1+1];
8     uint index1 = s_ifirst[L1];
```

```
9      uint index2 = s_ifirst[L1+1] + pair - s_pfirst[L1];
10     while (index2 >= s_ifirst[IdScan_MaxNumLayers]) {
11       index1 ++;
12       index2 -= nAbove;
13     }
14     ...
```

This change also brings improvement in all tests performed, particularly with the high luminosity sample on the Fermi card.

5.4.5 Using texture memory

If we access the spacepoint data using texture references, instead of standard global memory accesses, the data is stored in the texture cache. Each texture reference must first be bound to the appropriate address in memory. Reads are then performed using a specific texture function, in this case `tex1Dfetch`, to which is passed the texture reference and necessary offset.

Though this cache does not reduce the latency of future requests for the same data, it does eliminate the need a second global memory request, reducing the bandwidth strain. This frees global memory bandwidth to be used on other requests.

The results of using this method can be seen in fig. 5.6. There is a single case in which the performance improves — the high luminosity sample on the Tesla device. That there is a reduced performance on the Fermi device is expected. The Fermi device already caches global memory accesses, and the L1 and L2 caches have lower latency than the texture cache [6].

Interestingly, using the texture cache on Tesla gives better performance than the L1 and L2 caches on Fermi. However, the Fermi device is likely to remain the focus of any future investigations, so this optimisation was not used in the final code.

5.4.6 Triplet mode

In order to achieve satisfactorily accurate results with the high luminosity sample it is necessary to perform the calculation using triplets of spacepoints, as described in §3.1. This section of the code was added late in the project, and was essentially copied directly from the serial code, with little attempt at optimisation. The triplet code can be enabled or disabled by passing an appropriate value to the kernel.[4]

Introducing the triplet code does affect how certain arrays should best be placed in memory. The results with the triplet code can be seen in fig. 5.7. Triplet mode is unnecessary with low luminosity samples, so these were not tested.

[4] There are actually two different triplet enabled modes, but only triplet mode 1 — that described in §3 — will be used in this dissertation.

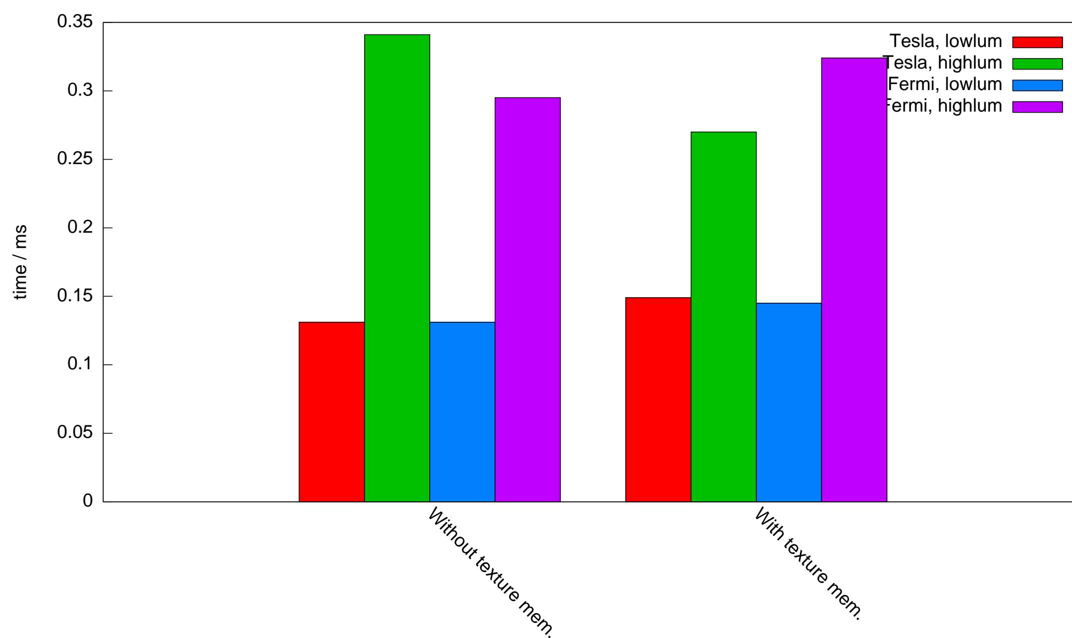

Figure 5.6: Execution time of findZ kernel when accessing global memory via textures references.

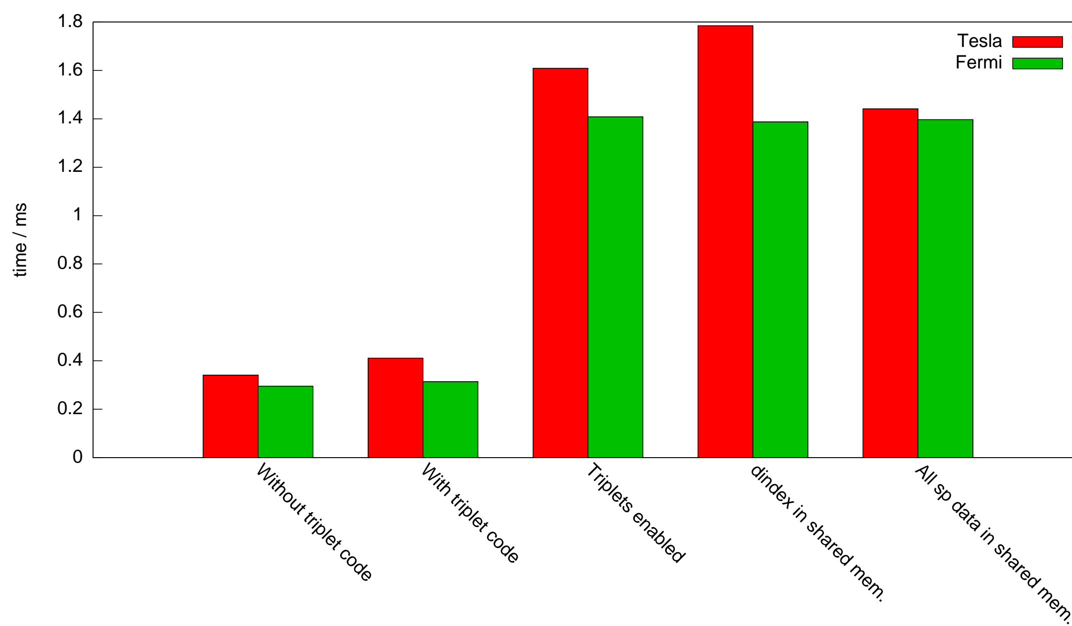

Figure 5.7: Execution times with high luminosity input after adding triplet code.

The initial drop in performance from adding triplet code is largely due to the fact that the code now reads the ϕ data into registers, which is not needed unless triplet mode is enabled. When the triplet mode is actually enabled the cost of the calculation is around 4x greater.

With `d_index` now being looped over considerably more times, it would seem very beneficial to move it into shared memory. However, fig. 5.7 shows that the performance actually decreases markedly on Tesla. With `d_index` stored in device memory, the shared memory usage of the kernel is a little under half of that available on a Tesla multiprocessor. This enables two blocks to execute simultaneously on each multiprocessor, which is very beneficial to the initialisation section of the kernel; otherwise not highly parallel. Moving `d_index` in shared memory raises the usage above half that available, so only a single thread block can execute. The Fermi device, with its larger shared memory, sees a slight benefit from this change.

Instead of storing the indices of the spacepoints in order, it is possible to copy the spacepoint coordinates into shared memory instead; again in order. Doing so slightly improves the performance of the triplet code on Tesla.

Chapter 6

Histogram Summation Kernel

Each thread block running the findZ kernel creates its own histogram in shared memory, but we wish to combine these into a single global histogram. In a CPU parallel architecture, such as OpenMP, each thread would add its result to a shared array. But to do this safely requires atomic instructions, otherwise race conditions may occur between threads. Unfortunately, the Tesla architecture do not implement floating-point atomic instructions, so it would not be possible to construct the `zHisto` this way. Additionally, on the Fermi architecture, the costs of atomic instructions in global memory will likely be prohibitive when hundreds of blocks are trying to write into the same array, but may be worthy of future investigation.

The alternative is for each block to write its histogram out to global memory separately, then reduce these down to single histogram later. Initially, we performed this reduction on the CPU, but in this chapter we discuss how best to perform the reduction on the GPU instead.

6.1 Measuring performance

Overall execution time is the clearest performance measure for this kernel, and the work is based around minimising this. However, summing histograms is a computationally cheap operation, so we expect this kernel to be memory bandwidth limited. It is therefore instructive also to measure the effective bandwidth of our kernel for comparison with the theoretical bandwidth of the device.

$$\text{effective bandwidth} = \frac{B_r + B_w}{10^9 \times t}\ \text{GB/s}$$

where B_r is the number of bytes read by the kernel, B_w the number of bytes written, and t the execution time for the kernel.[1]

[1] We have divided by 10^9 to convert to GB, where the pedant may demand 2^{30} be used instead. As we are only using this measure for comparison with the theoretical bandwidth, it makes no difference here.

Having a large effective bandwidth does not, however, imply that an kernel is at all efficient; it may simply be performing a large number of unnecessary global memory transfers, though we can only confirm this by examination of the algorithm. A low bandwidth, on the other hand, would indicate that the kernel is in need of modification.

The histogram summation execution time depends on the number of histogram bins and the number of ϕ slices, but on no other properties of the input. As these two values are similar for both of the samples, only the low luminosity sample was used for timing the summation.

6.2 Kernels

The histogram reduction on CPU is a substantial portion of the overall execution time. By porting this part of the code to GPU, we would hope to reduce this time considerably. More importantly, we dramatically reduce the amount of data that must be transferred off the device; by a factor of around 500, the number of ϕ slices.

All thread blocks must complete the findZ kernel before the histogram summation can be performed. As previously mentioned, no global synchronisation barrier exists in CUDA, but the completion of a kernel effectively acts as one, so the summation is implemented as a separate kernel.

6.2.1 First attempt

A very simple approach to performing the histogram reduction is seen in the following code listing.

```
1  __global__
2  void sumHistos_kernel(const slice* slices, uint nPhi, uint nBins)
3  {
4    for (uint j = 1; j < nPhi; j++) {
5      for (uint i = threadIdx.x; i < nBins; i += blockDim.x) {
6        slices[0].nHisto[i] += slices[j].nHisto[i];
7        slices[0].zHisto[i] += slices[j].zHisto[i];
8      }
9    }
10 }
```

`nPhi` is the number of histograms to be summed, and `nBins` the number of bins in each histogram.

The kernel is executed with 256 threads in each thread block as high occupancy gives the opportunity to hide some of the latency for global memory transfers. As the histogram bins are summed separately, we can achieve parallelism between threads without the need for a more complicated parallel algorithm.

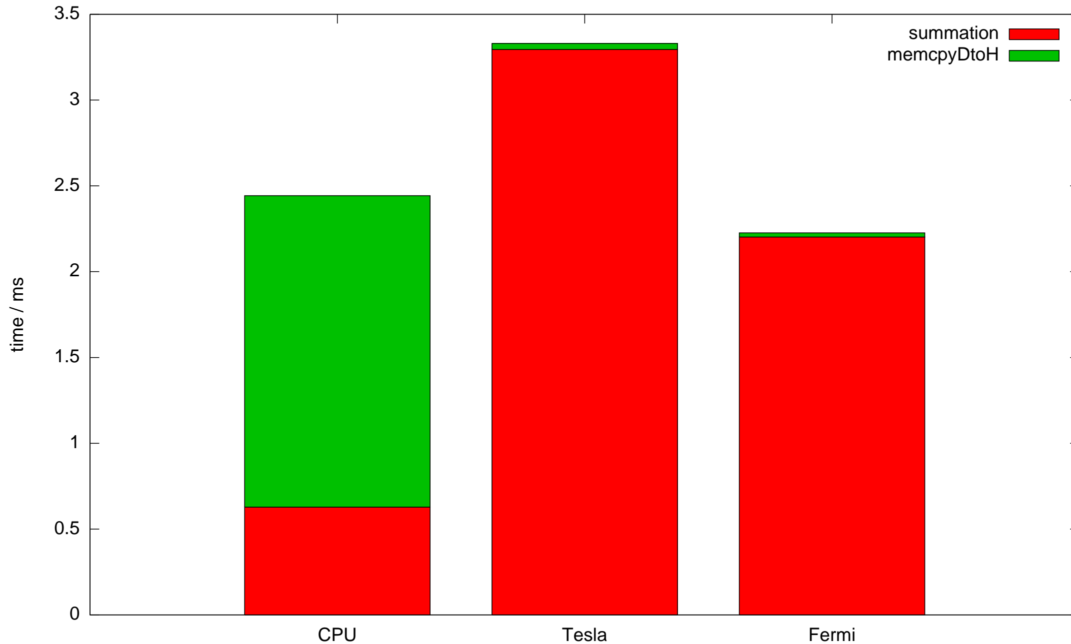

Figure 6.1: Execution times of the histogram summation and memory transfer, with summation performed on CPU and GPUs.

Looking at the results from this first attempt in fig. 6.1, we can see the substantial drop in time for the data transfer off the device to the host. The time for the summation itself increases with this first attempt, so we have not gained anything overall.

The performance of this kernel is much better on the Fermi device, due to the presence of the cache system. The histograms in which the results are being accumulated will be maintained in cache as they are continually being referenced, so the latency of reads and writes will be substantially reduced.

Even on the Fermi device, the bandwidth of around 4 GB/s very poor when compared to the theoretical bandwidth of 144 GB/s.

6.2.2 Performing summation in shared memory

On Tesla, the first kernel is performing many unnecessary reads and writes to / from global memory, as it is outputting all of the intermediate values and re-reading them on each iteration. We can eliminate this unnecessary cost by performing the summation in shared memory.

```
1  __global__
2  void sumHistos_kernel(const slice* slices, uint nPhi, uint nBins)
3  {
4    __shared__ uint s_nHisto[ZFinder_MaxNumZBins];
5    __shared__ real s_zHisto[ZFinder_MaxNumZBins];
```

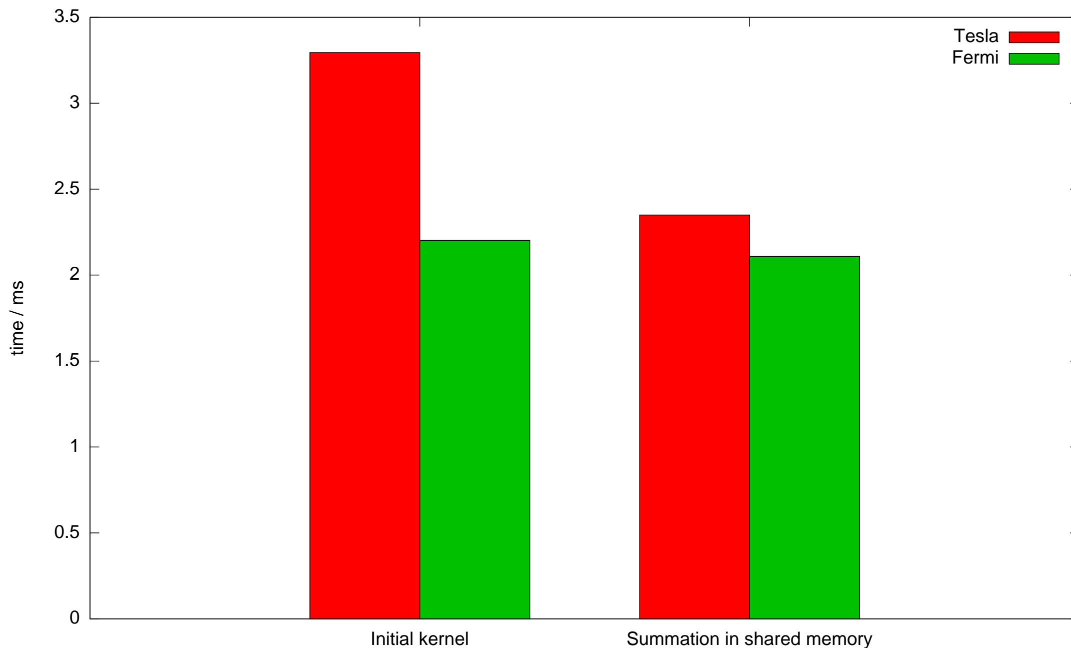

Figure 6.2: Execution time of sumHistos kernel utilising shared memory.

```
6
7     // zero histograms in shared memory
8     for (uint i = threadIdx.x; i < nBins; i += blockDim.x) {
9       s_nHisto[i] = 0;
10      s_zHisto[i] = 0;
11    }
12
13    // sum histograms in shared memory
14    for (uint j = 0; j < nPhi; j++) {
15      for (uint i = threadIdx.x; i < nBins; i += blockDim.x) {
16        s_nHisto[i] += slices[j].nHisto[i];
17        s_zHisto[i] += slices[j].zHisto[i];
18      }
19    }
20
21    // copy histograms to global memory
22    for (uint i = threadIdx.x; i < nBins; i += blockDim.x) {
23      slices[0].nHisto[i] = s_nHisto[i];
24      slices[0].zHisto[i] = s_zHisto[i];
25    }
26  }
```

We can see in fig. 6.2 that the use of shared memory leads to an appreciable improvement in performance on the Tesla device. As expected, on Fermi the improvement is more slight, as the cache was mostly disguising the problem already. The bandwidth is

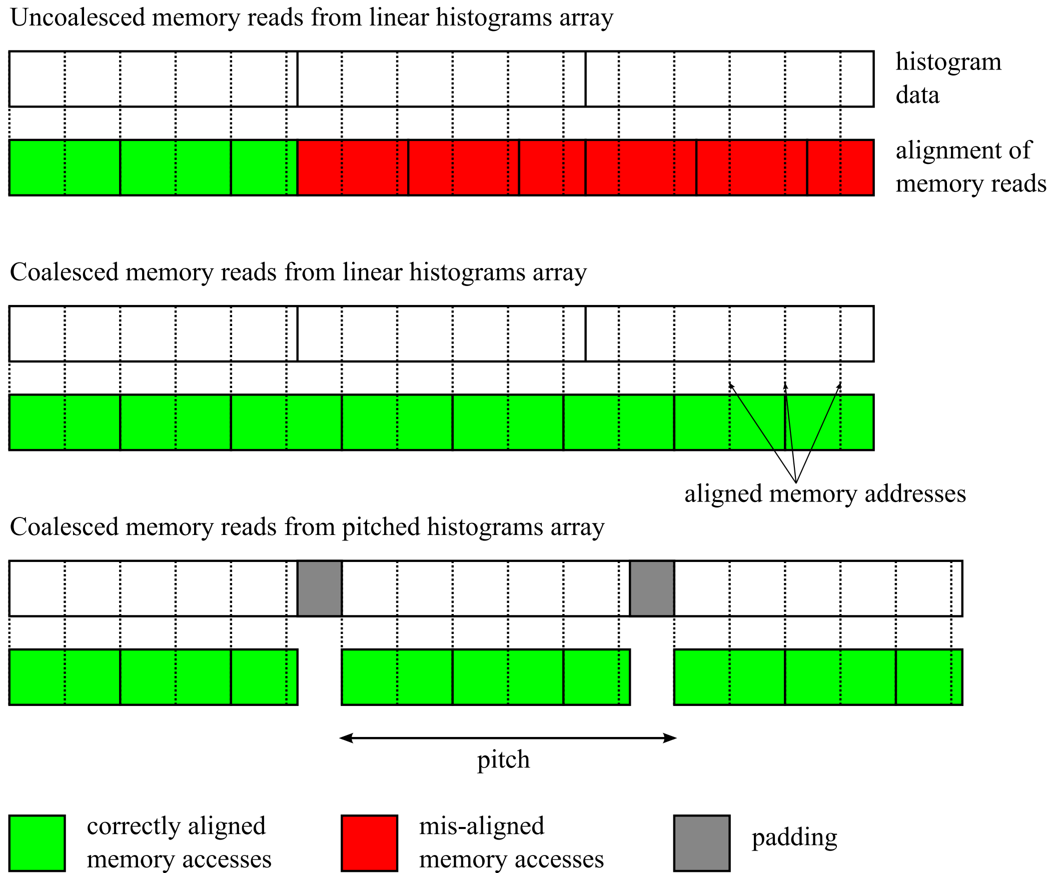

Figure 6.3: Schematic showing the alignment of memory accesses with different kernels.

lowered by this change, as there are many fewer global memory accesses.

6.2.3 Coalescing memory accesses

As the number of bins in our histogram will not, in general, be a multiple of 32, the vast majority of the global memory accesses will not be correctly coalesced — the first case displayed in fig. 6.3. Here we discuss three different methods for coalescing these reads; the results shown in fig. 6.4.

The following kernel ensures that all reads from global memory are coalesced, making use of the fact that the histograms for each slice are consecutive in memory – the second access pattern shown in fig. 6.3.

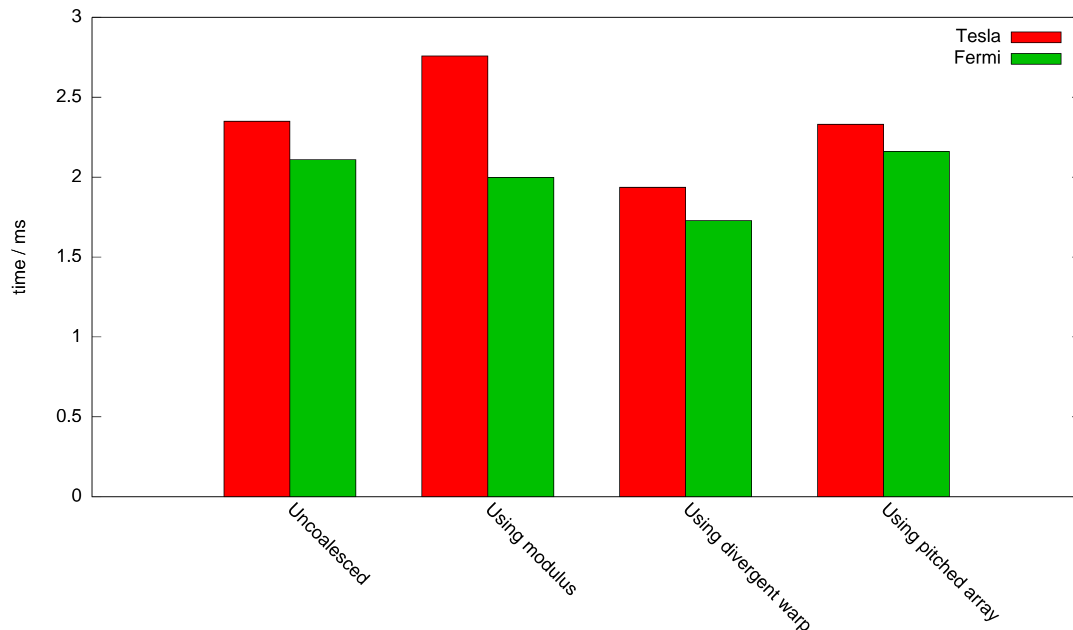

Figure 6.4: Execution time for histogram summation with different methods of coalescing global memory accesses.

```
14    ...
15    __syncthreads();
16
17    // sum histograms in shared memory
18    for (uint i = threadIdx.x; i < nPhi*nBins; i += blockDim.x) {
19      s_nHisto[i%nBins] += slices[0].nHisto[i];
20      s_zHisto[i%nBins] += slices[0].zHisto[i];
21    }
22    __syncthreads();
23
24    // copy histograms to global memory
25    ...
```

The beginning and end lines of code are omitted, but this sample could be inserted directly into the previous kernel at the line numbers specified on the left.

This kernel adds the additional overhead of a couple of thread barriers (on lines 15 and 22), as it is now possible for multiple threads to write to a single index of the shared arrays. However, this overhead cannot account for the reduction of performance witnessed on Tesla.

Perhaps surprising, the culprit is the integer modulus function (%), which is not supported directly in the GPU hardware. Both integer modulus and integer division require around 100 cycles to compute on Tesla, so should be avoided wherever possible. The results of this kernel suggest that integer modulus is better implemented on the newer

Fermi.

The next kernel maintains the same access pattern but removes this expensive modulus function, at the cost of introducing temporarily divergent warps. The warps may diverge for a single instruction, when `j` wraps around, as only some threads in the warp will execute `j -= nBins`.

```
16  ...
17  // sum histograms in shared memory
18  for (uint i = threadIdx.x, j = i; i < nPhi*nBins;
19       i += blockDim.x, j += blockDim.x) {
20    if (j >= nBins) j -= nBins;
21    s_nHisto[j] += slices[0].nHisto[i];
22    s_zHisto[j] += slices[0].zHisto[i];
23  }
24  ...
```

With this method, we see an improvement on both GPU devices. A warp can only diverge for a single instruction, so the cost of any divergence is low; much lower than the cost of the modulus function.

There is another method by which we can attain coalesced reads from global memory; making use of the `cudaMallocPitch` function. These allow us to allocate a two-dimensional array on the GPU, adding appropriate amounts of padding to each row to ensure that the start of each row is correctly aligned — the third case shown in fig. 6.3.

Using a pitched array will improve the performance of the findZ kernel marginally, as it will ensure coalesced writes on the final output. We can make this change seamlessly, with no alteration to either the findZ or sumHistos kernel, by setting the pointers in the slice structs appropriately for the pitched array.

Surprisingly, using a pitched array only gives a slight benefit to performance on Tesla. Why the improvement is so minor is unclear. The fact that the previous kernel was reading in a completely linear fashion maybe was allowing for further optimisations and / or memory prefetching to occur.

On Fermi, the performance is actually worse with a pitched array, on which reading from a simple linear array was well suited to the cached memory architecture, because no bandwidth was wasted.

Using pitched arrays does not, at this point, look particularly appealing, but becomes more so as further optimisations are applied, so we will persist with it. In particular, when performing the summation in multiple stages, pitched arrays allow for the intermediate sums to be written out in a coalesced form, whilst maintaining simple code.

The first improvement we can make is to move the read of the histogram pointers for the slice outside of the `i` loop. The compiler fails to do so and each requires a global memory read. As seen in fig. 6.5, this almost halves the time to execute on the Tesla card.

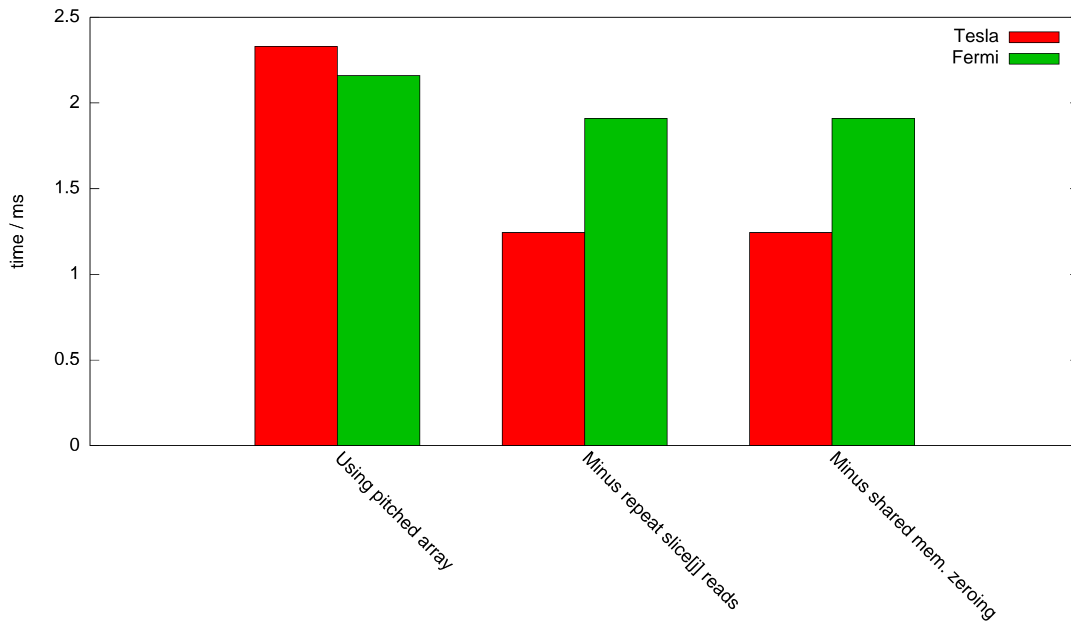

Figure 6.5: Execution time of sumHistos kernel with improvements to pitched array summation.

The improvement on Fermi is far more modest, as the caching system hides the cost of the re-read to some extent. Interestingly, the Fermi performance is now lower than that on the Tesla, showing that the cache system can be a mixed blessing. Upon a cache miss, an entire 128 byte cache line will be read from global memory, even though the threads only require a couple of pointers, wasting considerable bandwidth. Tesla would perform the read as a 32 byte request.

Another slight optimisation is to read the values of the first histogram directly into shared memory, rather than zeroing the memory first. The improvement though is virtually negligible.

Usually, using a pitched array adds a slight overhead due to the copy between two arrays of different pitch on the host and device, performed using the `cudaMemcpy2D` function. In this case though, we copy nothing into the arrays and only copy a single row off the device, so have no need for this function.

6.2.4 Multi-step summation

All of the summation kernels given so far suffer a major inefficiency; they can only run on a single thread block, and therefore do not come close to exploiting the potential computational power or memory bandwidth available on the GPU.

In order to execute the kernel on more than one thread block, it must be performed in multiple stages; again due to lack of global synchronisation.

```
6   ...
7   uint* myNHisto = slices[blockIdx.x].nHisto;
8   real* myZHisto = slices[blockIdx.x].zHisto;
9
10  for (uint i = threadIdx.x; i < nBins; i += blockDim.x) {
11    s_nHisto[i] = myNHisto[i];
12    s_zHisto[i] = myZHisto[i];
13  }
14
15  for (uint j = blockIdx.x+gridDim.x; j < nPhi; j += gridDim.x) {
16    uint* nHisto_j = slices[j].nHisto;
17    real* zHisto_j = slices[j].zHisto;
18
19    // sum histograms in shared memory
20    for (uint i = threadIdx.x; i < nBins; i += blockDim.x) {
21      s_nHisto[i] += nHisto_j[i];
22      s_zHisto[i] += zHisto_j[i];
23    }
24  }
25  ...
```

The blocks are allocated histograms in a cyclic manner, hence the `gridDim.x` increment in the `j` loop.

Executing the kernel as a two-stage reduction is performed as follows:

```
1 uint xblocks = uint(sqrt(nPhi));
2 sumHistos_kernel<<<xblocks, threads>>>(d_slices,    nPhi, nBins);
3 sumHistos_kernel<<<      1, threads>>>(d_slices, xblocks, nBins);
```

After the first stage, `xblocks` intermediate histogram sums will be output, so this value is passed in place of `nPhi` in the second stage.

In theory, the most efficient number of blocks for the first stage of a two-stage reduction is $\sqrt{\texttt{nPhi}}$.[2] In general, for a multiple stage reduction, we want to reduce to amount of data by the same fraction in each stage.

So, if we wished to reduce sixteen histograms in two stages, in the first stage, we use four blocks to reduce down to four histograms, then a single block in the final stage to reduce to the final result. This is shown in fig. 6.6. Ignoring overheads, we would have approximately halved the time for this reduction.

We can see in fig. 6.7 that there is a dramatic improvement in the execution speed for the summation; a ten-fold increase in performance by moving to two stages. Each additional stage adds further overhead from the kernel invocation and another set of writes storing and reads retrieving intermediate results to / from global memory, so

[2] Unless $\sqrt{\texttt{nPhi}}$ is greater than the number of blocks which can run simultaneously on the device, in which case the calculation is more complicated.

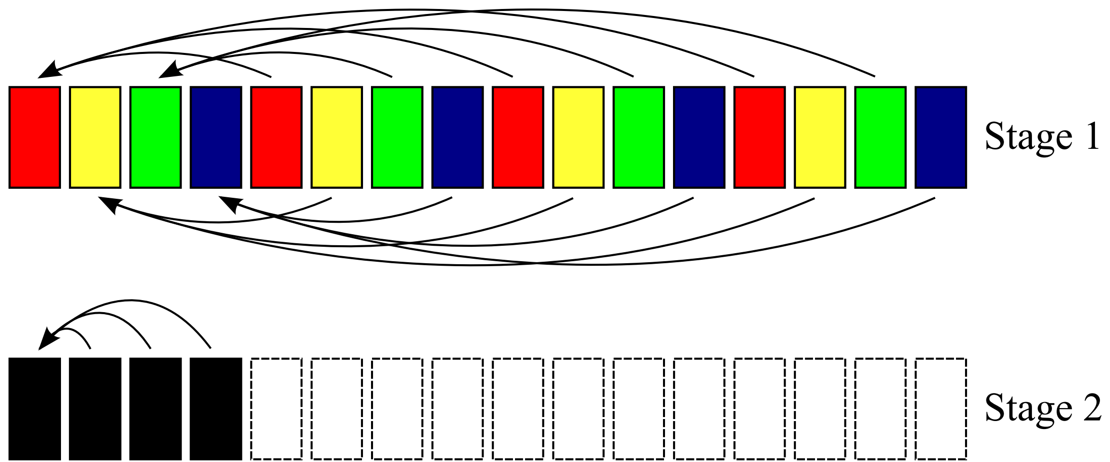

Figure 6.6: Schematic showing how a two-stage reduction is performed on 16 histograms.

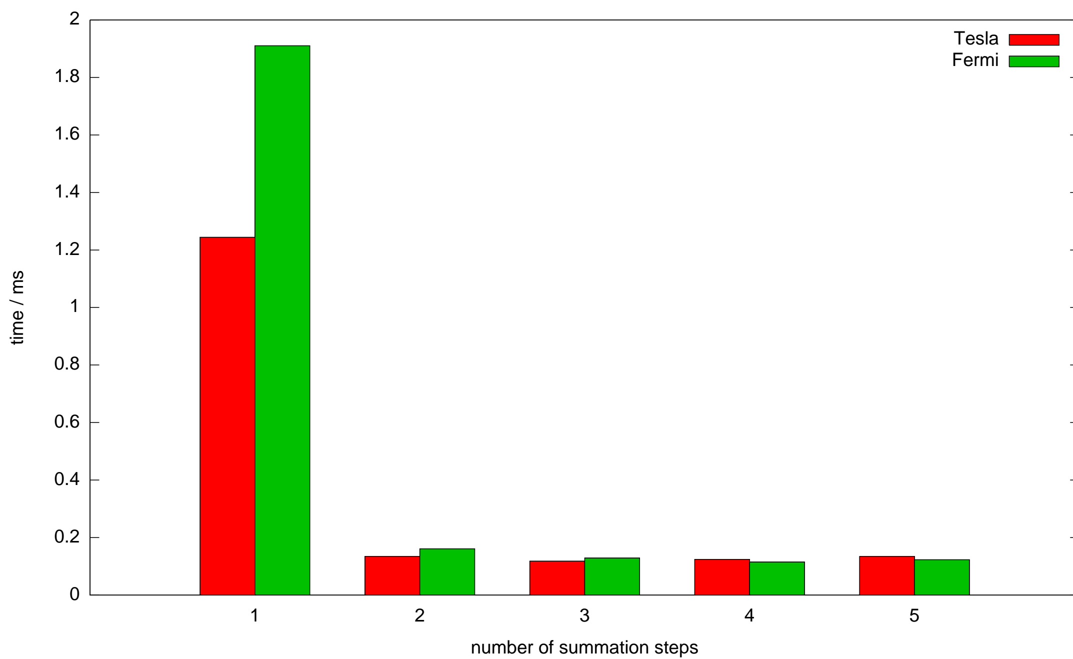

Figure 6.7: Execution time of summation time when performed in multiple steps.

there is little or nothing gained by moving to three or more stages. The advantage of additional stages is that we are doing more work in parallel for more of the summation, though this is still limited by the number of multiprocessors.

From these results alone, one would probably come to the conclusion that the optimum would be three or four stages. However, as further optimisations are applied, the overheads of each additional stage become more apparent. With hindsight, it is apparent that in the final version of the kernel, a two stage reduction is best, so for the sake of simplicity we will use only two stage reduction from this point onwards.

With a two stage summation, the bandwidth on both devices is around 20 GB/s; still well below the theoretical bandwidth, but a significant improvement.

6.2.5 Increasing the number of thread blocks

Even running the reduction in multiple stages, there may be insufficient thread blocks to occupy all of the GPU multiprocessors. This is clearly true of the final stage of the reduction, in which only a single thread block can ever execute.

We may increase the number of executing thread blocks by removing the loop over histogram bin, `i`, present in the previous kernel and instead run the kernel on a two-dimensional grid of thread blocks; each block in the y-direction of this grid performing a different iteration of this loop. The x-dimension of the grid is as before.

```
9     ...
10    uint i = blockIdx.y*blockDim.x + threadIdx.x; // bin number
11
12    if (i < nBins) {
13      s_nHisto[threadIdx.x] = myNHisto[i];
14      s_zHisto[threadIdx.x] = myZHisto[i];
15
16      // sum histograms in shared memory
17      for (uint j = blockIdx.x+gridDim.x; j < nPhi; j += gridDim.x) {
18        s_nHisto[threadIdx.x] += slices[j].nHisto[i];
19        s_zHisto[threadIdx.x] += slices[j].zHisto[i];
20      }
21
22      // copy to global memory
23      myNHisto[i] = s_nHisto[threadIdx.x];
24      myZHisto[i] = s_zHisto[threadIdx.x];
25    }
```

We execute the kernel on a two-dimensional grid as follows:

```
1 uint yblocks = (nBins+THREADS-1)/THREADS);
2
3 dim3 grid1(xblocks, yblocks);
4 dim3 grid2(      1, yblocks);
```

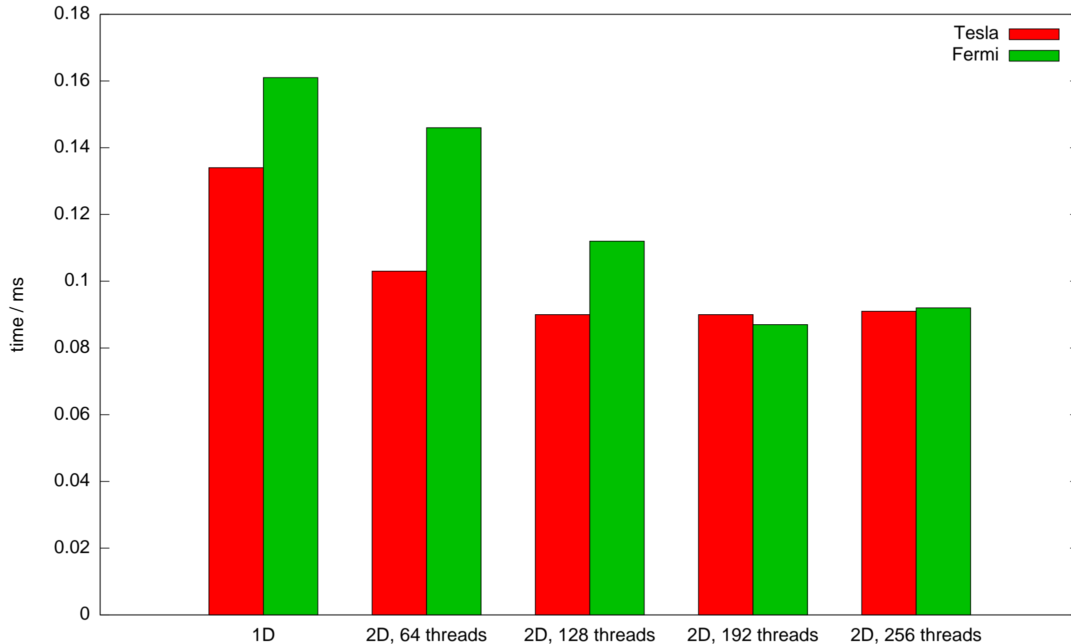

Figure 6.8: Execution time of summation performed on a 2D grid, with different numbers of threads per block.

```
5 sumHistos_kernel<<<grid1, THREADS>>>(d_slices,    nPhi, nBins);
6 sumHistos_kernel<<<grid2, THREADS>>>(d_slices, xblocks, nBins);
```

Where before each thread was performing the sum for multiple bins of the histogram, now each is responsible for only a single bin.

The choice of thread block size is now of greater consequence, as we must try to find the right balance between having too many small blocks in the first stage of the reduction and too few large in the final stage. We see in fig. 6.8 that 192 threads appears to provide the best compromise between these opposing aims, providing about a 30% improvement over the 1D grid. The bandwidth has now increased to around 35 GB/s per second on both devices.

6.2.6 Performing summation in registers

In the last incarnation in the kernel, we can see that each thread has a single element in each shared array dedicated to it. By storing the intermediate results in registers instead, we can remove many unnecessary instructions and the read / write overheads from shared memory. Though, as we see in fig. 6.9, the improvement from this change is negligible, particularly for the Tesla.

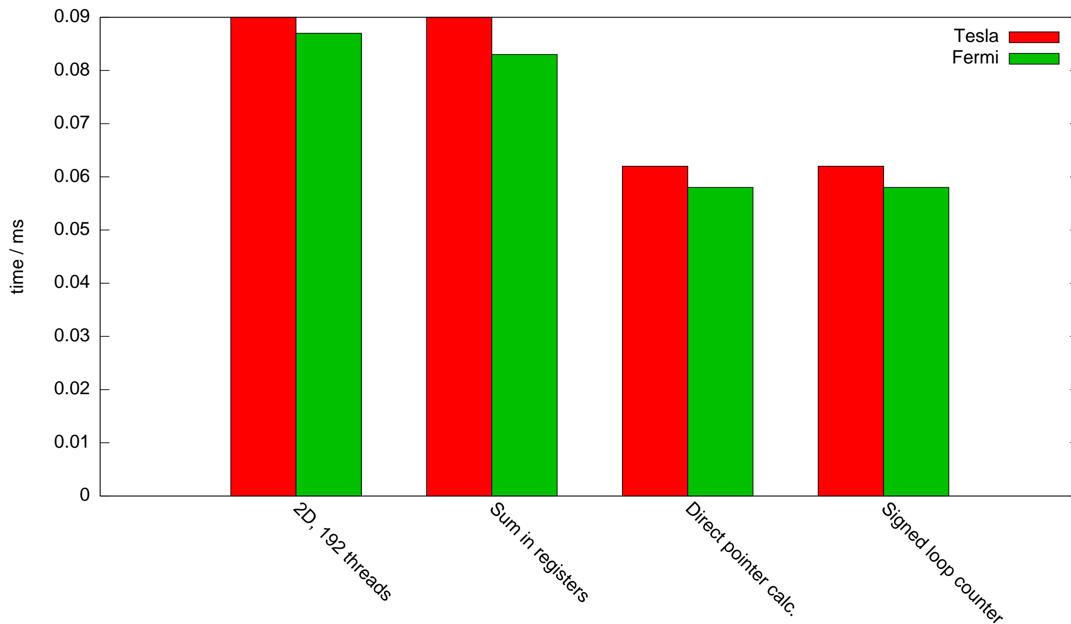

Figure 6.9: Execution time of summation after applying each of the final three optimisations applied.

6.2.7 Directly calculating histogram pointers

Whilst storing pointers to the each slice's histograms in the `slice` struct does make the code cleaner and more flexible, reading these pointers is adding an unnecessary global memory access at every loop iteration.

We can calculate the pointer for each slices histogram directly within the kernel:

```
15      ...
16      // sum histograms in registers
17      for (uint j = blockIdx.x+gridDim.x; j < nPhi; j += gridDim.x) {
18        nSum += ((uint*) (((char*) nHisto) + j*nPitch))[i];
19        zSum += ((real*) (((char*) zHisto) + j*zPitch))[i];
20      }
21      ...
```

The pointers to the two histograms, `nHisto` and `zHisto`, and their respective pitches, `nPitch` and `zPitch`, are passed as kernel arguments in place of the `slices` array.

When calculating the address of the histogram, it is necessary to cast to and from a **`char`** pointer as the pitch is in bytes.

This optimisation provides a further 30% improvement over the previous kernel, as seen in fig. 6.9.

6.2.8 Minor optimisations

The CUDA Best Practices Guide [5] suggests that it is preferable to use a signed integer for a loop counter, instead of an unsigned integer as has been used thus far. This is because the overflow behaviour of the signed integer is not strictly defined as it is for an unsigned integer. This allows the compiler to use further optimisations, such as strength reduction, within the body of a loop. In this case, however, the benefits were found to be completely negligible.

Similarly, loop unrolling was found to not improve, or degrade, performance. Though the contents of the loop are simple, they are still expensive due to the global memory read, so the cost of evaluating the condition on the **`for`** loop is negligible.

6.2.9 Final kernel

If we compare the performance of the final kernel with original CPU summation in fig. 6.10, we see a dramatic improvement in performance, particularly when combined with the reduction in data transfer off the device. Considered together, moving the summation onto the GPU results in a speed-up of around 25x.

The final kernel has a bandwidth of just over 50 GB/s on both devices; still below half of the theoretical maximum. In part, this is due to the relatively small amount of data. If the kernel is run on a RoI encompassing an entire $360°$ range, with 1800 slices, the bandwidth is increased by around 10%. Otherwise, it seems unlikely that the bandwidth can be increased much further, when one of the two stages in the summation is performed by only a few thread blocks.

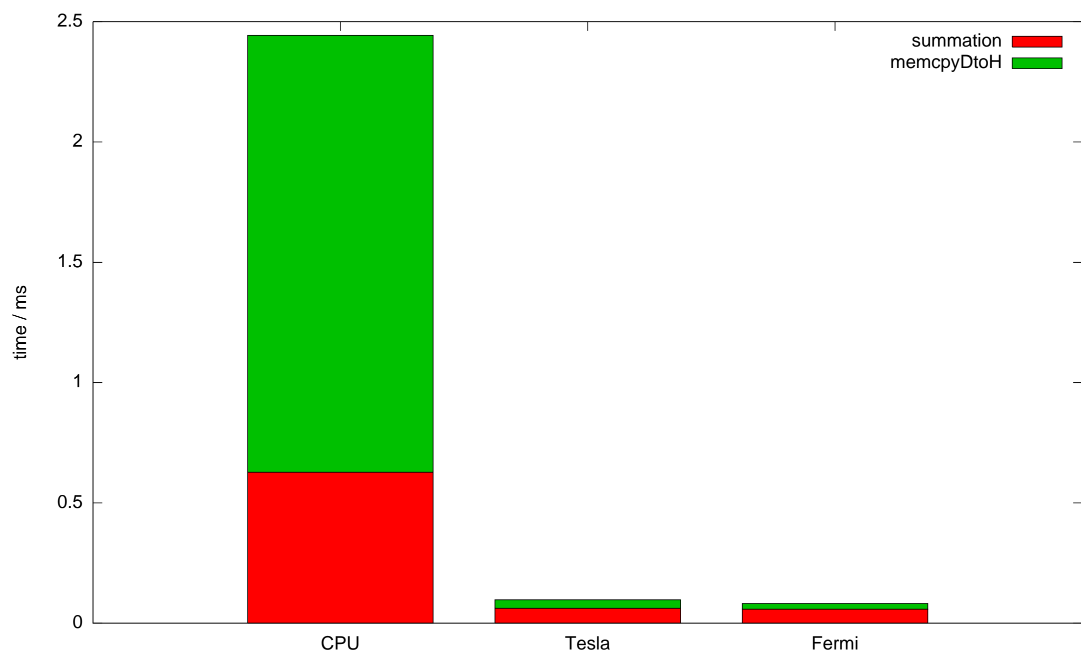

Figure 6.10: Final execution time of histogram summation and memory transfer, with summation performed on CPU and GPU.

Chapter 7

Wrapper Code

We complete our investigation by looking at the code surrounding and invoking the kernels developed in the previous two chapters.

7.1 Pre-allocating arrays

If we now look at the times for the *setup* and *cleanup*, we see this is dominating the overall cost; around 0.4 ms for each. It is in these sections that the arrays required by the algorithm, on both host and GPU device, are allocated and freed. The arrays are allocated to the minimum possible size on every call of the function.

We can drastically reduce this cost by instead moving the allocation of each array into the constructor for the `IDScanZFinderInternal` class and the memory release into the destructor. These are each called only once when the program is executed. In the real-world usage, with the program running continuously, the cost is effectively eliminated entirely.

As we cannot know in advance how large the arrays must be, it is necessary to allocate large blocks of memory to ensure they do not overflow. In practice, even the largest of these arrays need not exceed a few megabytes, which presents no problem on modern machines where both the host and GPU device possess gigabytes of memory.

7.2 Consolidating spacepoint data into single array

The size of the arrays containing the spacepoints data are relatively small, particularly at low luminosity. With these small arrays, the overhead from the `cudaMemcpy` function call is not negligible. By placing the spacepoint data consecutively in memory, we can copy this to the device in a single transfer and reduce this overhead.

This was done by allocating a single large array, `spData`. Pointers were then manually set at the appropriate point in this array for each of the four arrays of spacepoint data before these arrays are filled.

```
real* phi   = (real*) spData;
real* rho   = (real*) &(phi[nsp]);
real* zed   = (real*) &(rho[nsp]);
uint* inext = (uint*) &(zed[nsp]);
uint* lyr   = (uint*) &(inext[nsp]);
```

The **`real`** data arrays are placed first, as this ensures that there are no pointer alignment options if double precision floating-point arithmetic is used. The layer data is placed last, as this is not required on the GPU, and this way we can exclude it, whilst copying the first four arrays in a single transfer.

Consolidating the arrays results in a 35% improvement in the transfer time onto the device.

7.3 Overlapping CPU code and GPU code

A common optimisation method on more recent GPU devices is to take advantage of the ability to overlap communications between with host and device with the calculations on the device. The ability exists to hide the relatively low bandwidth of transfers between host and device, which is often a bottleneck, and is utilised through CUDA *streams*. Functions and kernels can be queued up to run on a specified stream, and will do so in order. Different streams though can execute in any order with relation to one another, or concurrently. On a Fermi device, different kernels can be run concurrently on the device.

The standard CUDA functions, such as `cudaMemcpy`, run synchronously, not returning control until the command has completed. Many though have an asynchronous counterpart, e.g. `cudaMemcpyAsync`, which takes a stream as an additional argument and returns immediately. Kernel launches can also run asynchronously when passed a stream.

In order to perform asynchronous memory transfers, the host array must be allocated in page-locked, a.k.a. pinned, memory.[1] This can be done using the `cudaMallocHost` function.

7.3.1 Streaming strategy

In many cases, an effective strategy is to divide the input data into pieces which can be operated on independently, with a different stream for each piece. But, the findZ kernel

[1] This is memory in a virtual memory system guaranteed not to be swapped onto disk.

requires all of the data from an RoI to be present on the GPU before commencing, so this strategy is not usable. Instead, we can overlap communications with calculations for other RoIs, running two parallel streams. In fact, our situation is well suited for overlapping, as the problem has been broken down into small independent pieces for us by the Level 1 trigger.

By overlapping correctly, we can ensure that there is always data ready on the device for processing and the GPU is continually occupied, maximising performance. Assuming that the GPU kernels are indeed the bottleneck, the costs of the memory transfers and any serial code are essentially eliminated, apart from the *warm-up* and *cool-down* periods at the beginning and end. With two streams, the calculations would progress as shown in fig. 7.1.[2]

Two copies of all arrays must be maintained; one for each stream. When the algorithm is called, it enqueues the memory transfers and kernel launches on the currently stream. The stream is then switched. As there are only two streams, the next stream is also the previous stream.

The code then stalls until the new stream is completed: when the results from the previous calculation have been retrieved. The peak is then found and returned. The algorithm, therefore returns the peak for the previous set of data, so a null set of data is used to retrieve the final set.

The results from overlapping can be seen with the other final results in the following chapter.

[2]Whilst the figure shows the findZ kernel being overlapped with the sumHistos kernel on the other stream, this is only possible on Fermi.

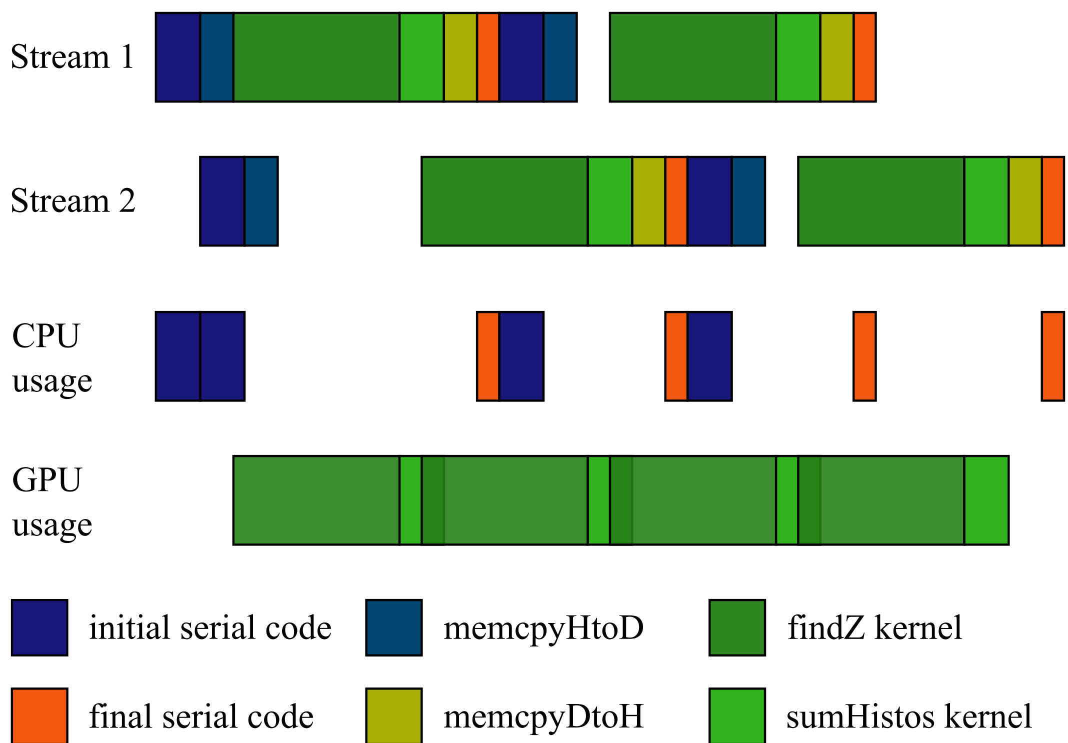

Figure 7.1: Schematic showing the progression of the calculation performed with two streams. Times for each part are very approximate.

Chapter 8

Discussion

The results of the final version of the code can be seen in fig. 8.1, and with overlapping enabled in fig. 8.2. The results show that this GPU code is unable to outperform the serial code on the low luminosity sample.

There is, however, a substantial speed-up shown on the high luminosity sample. With the non-overlapped code, the speed-up is in the range 9x–12x, with or without triplets being enabled. The triplet mode enabled results are shown in fig. 8.3.

With overlapping enabled, the results for high luminosity vary from 15x speed-up for the Tesla in triplet mode, up to 35x speed-up for the Fermi without triplets enabled. Probably the most interesting result, with regard to future development, is that for the Fermi with triplet results enabled. This speed-up here is 20x.

These results suggest that further investigations of porting ATLAS trigger code to GPU are very much worthwhile. Whilst a simple OpenMP parallelisation on a modern, quad-core processor could achieve up to 4x speed-up, 20x speed-up is unlikely to be achievable on any single CPU currently in production. This justifies the added complexity of the code, and probably also the expense of purchasing additional hardware.

Running the overlapping version of the code would certainly require modification to the existing framework, but the problem is certainly not insurmountable.

8.1 Future work

There are many avenues for potential improvement to this GPU port for which there was insufficient time to investigate. Here are a few which may be worth investing future effort in.

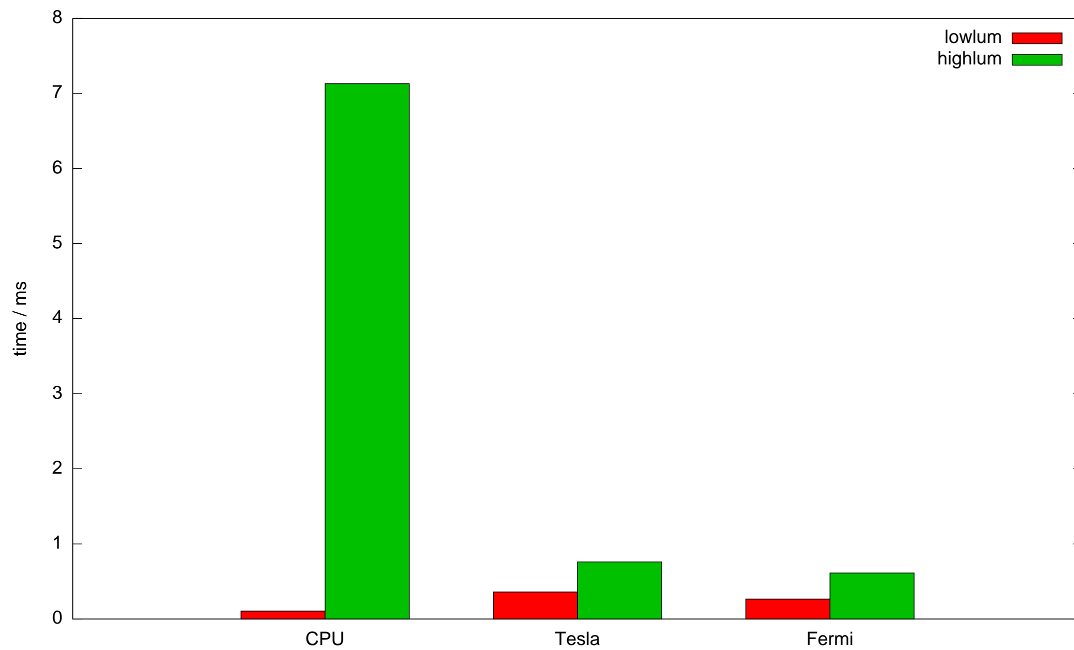

Figure 8.1: Final execution time for non-overlapping GPU code compared to original CPU code.

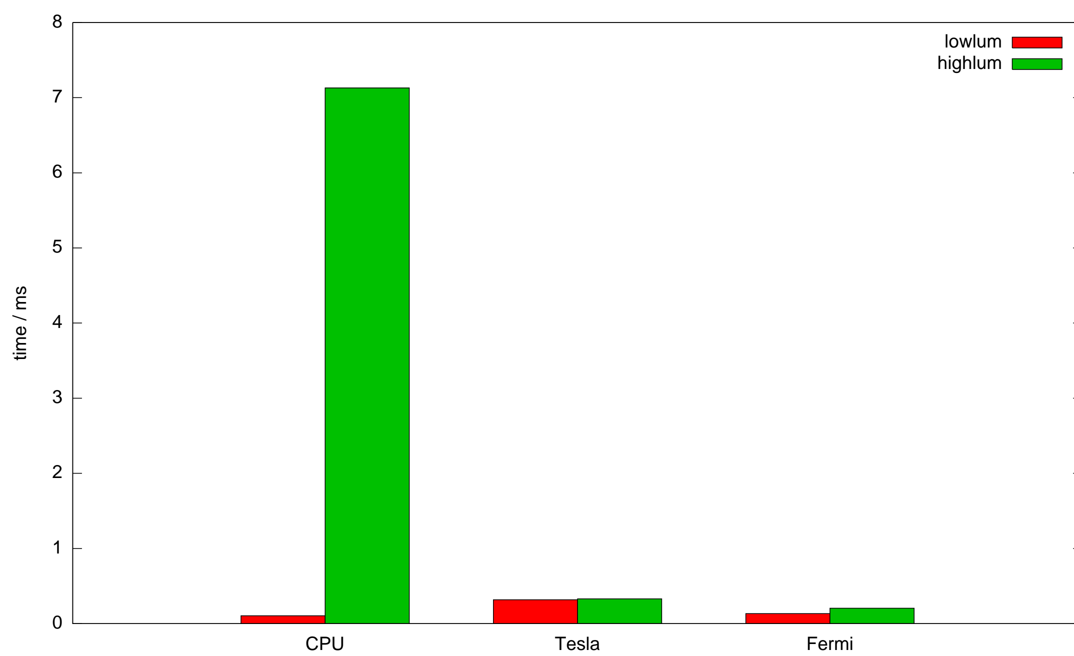

Figure 8.2: Final execution time for overlapping GPU code compared to original CPU code.

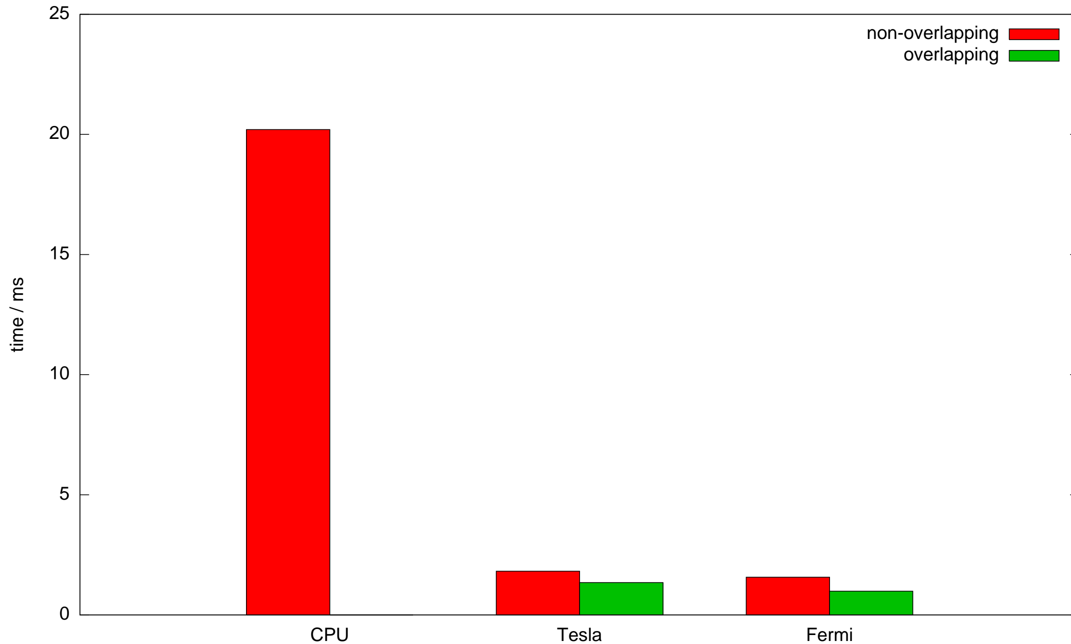

Figure 8.3: Final execution time for GPU code compared to original CPU code, for the high luminosity sample with triplet mode enabled.

8.1.1 Load only two slices per block

In the final code, each block loads data from three slices, and calculates pairs from the central slice going up, left and right. An identical calculation could be performed, whilst loading only two slices: pairs from the left slice going up and right, and pairs from the right slice going left only. This modification would complicate the logic of the kernel, but would reduce the bandwidth requirements of global memory accesses, so may be beneficial.

8.1.2 Using CUDA vector types

The findZ kernel currently issues separate requests for each coordinate of a spacepoint. CUDA includes built-in vector types, such as `float4`, which would allow multiple coordinates to be retrieved in a single request. If the coordinates of a spacepoints were contained in such a vector type, the burden on global memory would be reduced.

8.1.3 Doing more work on GPU

A general principle of GPGPU work is to implement as much of the code as possible on the GPU, as this reduces unnecessary data transfers. There are two areas of serial code with potential to be ported, currently book-ending the GPU code.

The first of these is the spacepoint 'sorting' code, which if ported, would eliminate the need to transfer the array of `slice` structs. This could trivially be ported to GPU, but would make heavy use of atomic instructions in global memory, so might be prohibitively expensive. It would also require transfer of the spacepoint layer data to GPU, which does not occur currently, so might actually increase transfer size with very high spacepoint numbers.

The second is finding the peak of the histogram. If this were ported, only the vertex itself would need to be transferred off the device, rather than the summed histograms, so this may prove beneficial.

8.1.4 Doing more work on CPU

If, instead, we consider the overlapped version of the code, the situation is reversed. Whilst the kernel computation remains the bottleneck, there is nothing to be gained by porting more code to GPU. The memory transfers are performed at virtually no cost, and the CPU is not being fully utilised. Therefore, it is possibly beneficial to move work onto the CPU, even if it requires copying more data across.

One possibility might be to do an actual sort of the spacepoints into slices and layers on the CPU, before the transfer, which would not increase the size of the data transfer. Another alternative might be to perform some of the initialisation section of the findZ kernel on CPU, and copy this across as an additional part of the `slice` struct. If this was done, one could also identify slices with no pairs to calculate, reducing the number of threads blocks launched for the findZ kernel.

Chapter 9

Conclusions

Porting the z-finder algorithm to GPU using C for CUDA was found to be an effective method of improving performance with high luminosity samples from a simulation of physics events after the upgrade of the ATLAS experiment. No speed-up was attained with the low luminosity sample.

The non-overlapping version of the ported code resulted in a speed-up of around 10x. Using CUDA streams to overlap calculations for different regions of interest was found to improve the performance yet further, with up to **35x** speed-up. **20x** speed-up was typical of the overlapping version of the code.

The new Fermi hardware represents a significant departure from the previous CUDA architecture. Optimisations which improved performance on the Tesla architecture often degraded performance on the Fermi, and vice versa. To achieve optimum performance, the code would need to be targeted at one platform or the other.

The findZ kernel shows that to fully utilise the processing cores, threads must have as much parallel work to perform as possible. Where this is not possible, executing multiple blocks on a single multiprocessor can have the same effect. Similarly, the sumHistos kernel demonstrates that having insufficient blocks to occupy all multiprocessors will result in lost performance.

The risks of divergent warps in CUDA are often overstated and, provided the divergence only lasts a few instructions, they can offer a good alternative to some expensive operations on the GPU, including integer modulus and division.

Having achieved this in a few months of work, I would recommend that further effort is invested by the ATLAS team in investigating methods of porting more of the ATLAS trigger onto GPU.

Bibliography

[1] ATLAS fact sheets. http://atlas.ch/fact_sheets.html.

[2] ATLAS photos. http://atlas.ch/photos.

[3] CUDA zone. http://www.nvidia.com/object/cuda_home.html.

[4] ATLAS Collaboration. ATLAS: technical proposal for a General-Purpose pp experiment at the LHC. *CERN/LHCC*, 94(43), 1994.

[5] NVIDIA Corporation. *CUDA Best Practices Guide*, 3.1 edition.

[6] NVIDIA Corporation. *CUDA Programming Guide*, 3.1 edition.

[7] NVIDIA Corporation. *Fermi Architecture Whitepaper*.

[8] A. Gesualdi Mello et al. Overview of the high-level trigger electron and photon selection for the atlas experiment at the lhc. *Nuclear Science, IEEE Transactions on*, 53(5):2839 –2843, October 2006.

[9] N. Konstantinidis and H. Drevermann. Determination of the z position of primary interactions in ATLAS prior to track reconstruction. *ATLAS-DAQ-2002-014*, 2002.

[10] M. Sutton. Tracking at level 2 for the ATLAS high level trigger. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 582(3):761–765, December 2007.