

Algorithm Acceleration from GPGPUs for the ATLAS Upgrade

This article has been downloaded from IOPscience. Please scroll down to see the full text article.

2011 J. Phys.: Conf. Ser. 331 022031

(<http://iopscience.iop.org/1742-6596/331/2/022031>)

View [the table of contents for this issue](#), or go to the [journal homepage](#) for more

Download details:

IP Address: 95.172.225.198

The article was downloaded on 23/01/2012 at 08:02

Please note that [terms and conditions apply](#).

Algorithm Acceleration from GPGPUs for the ATLAS Upgrade

P.J. Clark¹, C. Jones¹, D. Emeilyanov², M. Rovatsou¹, A. Washbrook¹ on behalf of the ATLAS collaboration

¹ SUPA, School of Physics and Astronomy, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, United Kingdom

² Particle Physics Department, STFC Rutherford Appleton Laboratory, Harwell Science and Innovation Campus, Didcot, Oxfordshire, OX11 0QX, United Kingdom

E-mail: awashbro@ph.ed.ac.uk

Abstract. Feasibility studies into the use of General Purpose GPUs have been performed on two key algorithms in the ATLAS High Level Trigger. A GPGPU based version of the Z-finder vertex finding algorithm resulted in over 35 times speed-up over serial CPU execution in the best case scenario, whilst a speed-up of over 5 times was observed from a GPGPU-based Kalman filter track finder routine. The approaches taken for converting these algorithms for execution on GPU devices is described.

1. The ATLAS trigger

At full design performance, the Large Hadron Collider will achieve proton-proton collisions at a luminosity of $10^{34}\text{cm}^{-2}\text{s}^{-1}$ with a beam crossing rate of 40 MHz. To address the amount of filtering needed for this large volume of data, the ATLAS trigger rejects a large proportion of collision events at the first stage of processing before providing more detailed reconstruction of the smaller sample of the events of interest. Overall, the current ATLAS trigger reduces the total data rate by a factor of around 200,000.

The trigger operates in a three-tier hierarchy. The Level 1 trigger runs on custom built hardware whilst the Level 2 and the third level Event Filter, referred to collectively as the High Level trigger, are software triggers that run on a dedicated computing facility located close to the detector. The Level 2 trigger takes input detector data in parallel corresponding to different Regions of Interest (RoIs) in a collision event and consists of dedicated algorithms for event reconstruction including particle track reconstruction. The Event Filter has access to the full event information before the event is potentially stored offline for further processing and distributed worldwide via the LHC computing grid. A more detailed description of the trigger design and operation can be found elsewhere [1, 2].

The ATLAS trigger has so far been successful in dealing with data rates at the current run luminosity produced by the LHC. However a possible upgrade of the LHC and ATLAS will pose greater challenges, and emerging computational techniques such as GPU computing could be used in the trigger software in order to provide accelerated computation for significantly higher data rates. The studies into using GPUs for the ATLAS trigger focus on two routines contained

in the Level 2 trigger; the Z-finder vertex finder routine and the Kalman filter track fitter used for track reconstruction.

2. General Purpose GPU programming

General purpose computing on graphics processing units (GPGPU) is a relatively recent development in High Performance Computing. Graphics processing is a computationally intensive, highly parallel operation, and GPUs have been designed to be highly specialised for this task, sacrificing memory caching and sophisticated flow control in favour of floating-point performance. Modern GPUs easily surpass the capabilities of CPUs by maximising throughput of computations over vast volumes of data, enabling programs to fully utilise the hundreds of processing cores and high memory bandwidth available on the GPU.

There are several APIs and SDKs available which can be used to perform GPGPU programming. For the feasibility studies presented here CUDA [3] was chosen as it was found to be the most popular and mature of the solutions available.

In the CUDA model, CPUs are “hosts” responsible for flow control whilst GPUs are used as compute devices. A program is partitioned into sub-tasks executed in parallel by using kernels. A kernel is a function executed in parallel by a number of threads on the GPU device. Threads are arranged into thread blocks which are executed together on a single multiprocessor. Thread execution can then be synchronised by a barrier which allows for cooperation between threads in a single block. Before a kernel can be executed all data manipulated by the kernel must be explicitly transferred onto the GPU device. Afterwards, the result must be retrieved in the same fashion.

GPU devices contain many different types of memory, each with their own properties. The two areas of GPU memory most of interest here are global memory and shared memory. Global memory is the main memory storage on the GPU. This type of memory typically has an access latency of several hundred clock cycles, so it is important to avoid unnecessary read and write calls. Shared memory is located on each multiprocessor enabling efficient data sharing between threads in the same block. Although the memory size is much smaller than global memory, the latency of shared memory access is considerably lower than that of uncached global memory access.

The properties of the GPU devices used for performance testing in this study are shown in Table 1. The CUDA compute capability value [3] is also shown for each device. This value is used to describe the GPU architecture, the global and shared memory layout and how the device memory is accessed.

Table 1. Properties of GPU devices used for performance studies.

Properties	Tesla C1060 (Tesla)	Tesla C2050 (Fermi)
CUDA Capability	1.3	2.0
Global Memory	4.3GB	2.8GB
Multiprocessors	30	14
Cores	240	448
Threads/block	512	1024
Concurrent kernels	No	Yes
ECC support	No	Yes

3. Z Finder algorithm

The purpose of the Z-finder algorithm contained in the Level 2 trigger is to locate the z-coordinate along the direction of the LHC beam, z_0 , to an accuracy of 1mm. An accurate evaluation is of interest as this will improve performance and reduce execution time of the track reconstruction algorithms run later in the trigger.

The algorithm calculates the z-coordinate of the intercept with the z-axis of the trajectory through pairs of “spacepoints” - the coordinates of clusters of detector hits - in different layers of the ATLAS Inner Detector (see Figure 1). This value is inserted into a histogram and the calculation is repeated for all possible spacepoint pairings constrained within a single RoI (see Figure 2). The Z-finder algorithm is described in more detail elsewhere [4].

The amount of calculations can be significantly reduced by only considering pairs constrained within thin slices in the ϕ direction (the azimuthal angle, perpendicular to the beam direction). This optimisation step can be justified because the tracks of interest have a higher transverse momentum and will consequently bend little in the magnetic field of the ATLAS detector. The constituent spacepoints of the track will therefore be contained within an individual “ ϕ slice”. This logic is already included in the Z-finder algorithm but at present each ϕ slice is processed sequentially. Since the ϕ slices can be treated independently we can consider processing each ϕ slice in parallel. To this end, the use of GPUs to improve the performance of this algorithm appears to be an ideal candidate.

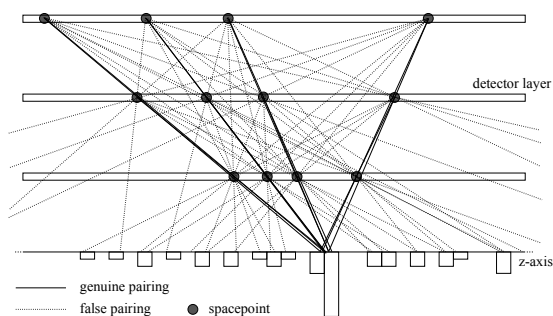


Figure 1. Schematic showing how z-vertices are calculated using pairs of particles within a ϕ slice.

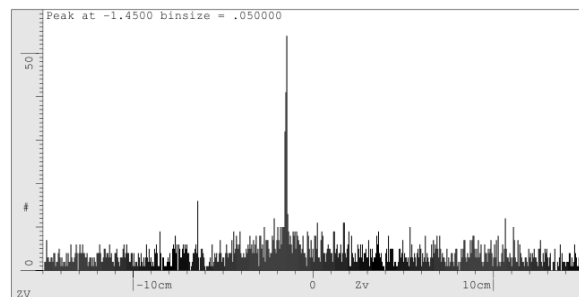


Figure 2. A sample Z-finder histogram for an electron candidate [4].

3.1. Test case and performance measurement

A standalone version of the algorithm was used to form a GPU-based test case. Two different simulated data samples were used for benchmarking the code: a lower luminosity sample ($10^{32}\text{cm}^{-2}\text{s}^{-1}$), representative of the data rate currently recorded by ATLAS, and a higher luminosity sample ($10^{34}\text{cm}^{-2}\text{s}^{-1}$), representative of the data received by the trigger at full LHC design performance. Relative performance was measured by extracting the execution time for each section of code, which was then averaged over a number of input events. Kernels were timed using the built-in CUDA event timer, which provided greater precision and consistency than the CPU timer.

3.2. Histogram Construction Kernel

The first approach considered was to allocate a single GPU thread to process spacepoints in each ϕ slice. This strategy was not found to be efficient since the typical number of ϕ slices was much lower than the total number of threads available, leading to under utilisation of the GPU

device. In addition, the number of spacepoints contained in different slices can vary by a large amount resulting in an unbalanced thread workload.

Instead, a thread *block* was allocated for each ϕ slice. To get close to exploiting the full computing power of the GPU device, the algorithm must be run on multiple threads within each thread block. This was done by providing additional logic in the algorithm to pre-allocate a pair of spacepoints to each thread in a block to enable the z_0 calculation for each spacepoint pairing to be calculated in parallel.

Ideally, the result from each thread block would be then aggregated into a single histogram located in global memory. This was not found to be desirable because frequent data transfer to and from global memory incurred a high latency penalty. Given this restriction, the best solution found was to store a separate histogram for each ϕ slice into the shared memory of each thread block. At the end of each thread block execution the histogram was copied to global memory. This step reduced the execution time considerably [5].

3.3. Histogram Summation Kernel

To complete the Z-finder algorithm conversion the set of histograms created by the histogram construction kernel must be combined into a single histogram before a final z_0 can be determined. Initially, this combination step was performed on the CPU but it was found to contribute to a substantial portion of the overall execution time. However, since the data from the previous kernel already resides on the GPU device a second kernel was implemented with the purpose of combining the data into a single global histogram. This significantly reduced the amount of data to be transferred off the device.

3.4. Overlapping Kernel Execution

A common optimisation method on more recent GPU devices is to take advantage of the ability to overlap data transfer between host and device with kernel execution. Different streams can then be executed concurrently which will result in GPU being continually occupied, thereby improving performance. This test case was well suited for overlapping execution of kernels, as the data to process has already been broken down into independent RoIs. When the algorithm was called, device transfers and kernel calls for one RoI were launched before switching to a second stream. The active streams were then toggled to minimise data transfer latency.

3.5. Results

The timing results are shown in Figure 3. Timing values are given for execution on GPUs based on the Tesla and Fermi architectures (i.e. GPUs with CUDA compute capability of 1.3 and 2.0 respectively) with and without concurrent execution using CUDA streams. For comparison, Figure 3 includes the algorithm timing when executed on a single CPU.

The results indicate that the GPU-based code is unable to outperform the serial code on the low luminosity sample. At low luminosities, the spacepoint occupancy for a given RoI tends to be lower resulting a much quicker execution of the original Z-finder algorithm. This is in contrast to the GPU-based code which has a small execution overhead, independent of the amount of spacepoints to be processed, primarily due to data transfer between the host and the GPU device.

Algorithm speed-up is therefore only observed after a minimum threshold of spacepoint occupancy. This was observed in the high luminosity sample where a substantial speed-up is found. Without the use of overlapping kernels, the speed-up is in the range of 9 to 12 times faster than the CPU-based code. With overlapping kernels, the results for high luminosity were 35 times faster on the Fermi GPU.

Note that the results from this study do not take into account all the running modes of the Z-finder algorithm currently in production. For example, triplets of spacepoints (as opposed to

pairs of spacepoints) can be used to derive a cleaner signature for z_0 in the histogram. With this mode enabled, a 20 times speed-up was achieved on a Fermi GPU without further optimisation.

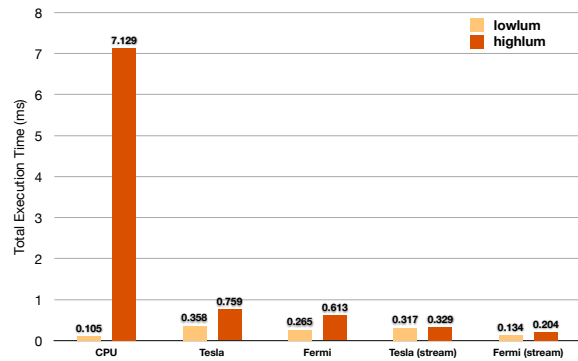


Figure 3. Timing results for the Z-finder test case for execution on a single CPU, and for Tesla and Fermi based GPUs. Results are also shown for the use of CUDA streams.

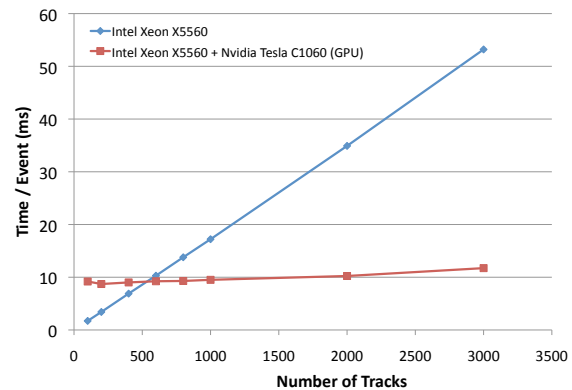


Figure 4. Timing results from the Kalman filter test case.

4. Kalman filter

The ATLAS High Level trigger uses a Kalman filter based technique as part of the track reconstruction process for each event. A track is reconstructed by measurements collected from each sub-detector which are fed into the filtering process. A detailed description of the Kalman filter technique used in the ATLAS Trigger can be found elsewhere [1].

Fast track reconstruction in the Level 2 trigger relies on the performance of the Kalman filter. A possible acceleration of the fitting function was considered by performing a track fit per GPU thread. Note that this aimed at increasing the throughput of the overall fitting process rather than accelerating the fitting process of a single candidate track. This approach bears many advantages, since there are no dependencies among the tracks in a given event. In addition, using GPUs for Kalman filter acceleration has been achieved successfully for other experiments [6].

4.1. CUDA C++ limitations

For timing studies to be performed it is necessary to convert the main Kalman filter fitting function into a CUDA kernel. However, this cannot be trivially migrated for GPU execution at this present time because the complete set of C++ features is not fully supported yet by CUDA.

In particular, a CUDA kernel is not able to dynamically allocate (or de-allocate) memory using `new` and `delete` operators and so all memory management must be done before the invocation of the kernel. Secondly, all track state information is represented by a multiple inheritance structure that cannot be referenced from within the scope of the kernel. Although there are some possibilities in CUDA to replicate inheritance structures the complexity of the fitting code meant that conversion was not possible.

4.2. Kalman filter Implementation

Given these limitations it was decided to re-implement a standalone version of the fitting code into C before a test kernel was written. The first implementation in C followed the original design of the Kalman Fitter code and retained its structure as closely as possible. Class elements were

stored in C structs and class functions were converted to C functions which retained the same functionality. The inheritance structure was also eliminated for simplicity.

It was immediately found that the volume of data to be allocated in GPU memory was extensive due to the lack of dynamic memory allocation available [7]. This was coupled with the requirement that data from intermediate steps of the Kalman filter have to be retained, and that the total memory required will increase linearly with the number of tracks.

A second C implementation of the fitting code was able to reduce the memory allocation needed by using CUDA built-in vector types to compact only the data required for the fitting process. A move to single precision variables to store all the data was also performed to aid memory allocation and performance.

4.3. Results

The performance of the GPU-based code was tested using simulated events with an increasing number of tracks. The timing results of the Kalman filter algorithm for serial CPU execution and for a Tesla GPU are shown in Figure 4. As expected the CPU execution of the Kalman filter scales linearly with an increasing amount of tracks. However for the GPU-based code, it is observed that the execution time is similar independent of the number of tracks processed, due to tracks being processed in parallel. At 3000 tracks per event - which is typically expected at higher luminosities - over 5 times speed-up over serial execution is observed.

Typically 60 million floating point operations are required to process an event containing 3000 tracks. This translates to a throughput value of 5.5 GFLOPS for the GPU-based code when considering the execution time shown in Figure 4. Note that this value is well below the theoretical peak performance of a typical GPU due to code performance being primarily bound by memory throughput. The performance could be improved by optimising the use of the faster shared memory on the GPU device.

5. Conclusions

GPU feasibility studies performed on two key algorithms in the ATLAS High Level trigger have resulted in a significant speed-up over serial CPU execution. A speed-up of over 35 times was found for the Z-finder routine whilst a speed-up of 5 times was observed for the Kalman filter for events containing 3000 tracks.

These results suggest that further investigations into migrating key sections of the ATLAS code for GPU execution are very much worthwhile. In principle, a GPU-based approach could also be used to process different sections of the detector in parallel to allow tracking in the complete detector to be performed within the same processing time as of a single ROI at present. This performance benefit may justify the added complexity of the code in advance of a future LHC and ATLAS upgrade activities.

References

- [1] Jenni P, Nessi M, Nordberg M and Smith K 2003 ATLAS high-level trigger, data-acquisition and controls: Technical Design Report ATLAS-TDR-016
- [2] Sutton M 2007 Tracking at Level 2 for the ATLAS High Level Trigger *Nucl. Instrum. Meth. A* **582(3)** 761-5
- [3] NVIDIA 2010 CUDA C Programming Guide (version 3.1) http://developer.nvidia.com/object/cuda_3_1_downloads.html
- [4] Konstantinidis N and Drevermann H 2002 Determination of the z position of primary interactions in ATLAS prior to track reconstruction ATLAS-DAQ-2002-014
- [5] Jones C 2010 University of Edinburgh MSc Thesis <https://twiki.cern.ch/twiki/bin/view/Main/AtlasEdinburghGPUComputing>
- [6] Bach M, Gorbunov S, Kisel I, Lindenstruth V and Keschull U 2008 Porting a Kalman filter based track fit to NVIDIA CUDA FAIR-EXPERIMENTS-38
- [7] Rovatsou M 2010 University of Edinburgh MSc Thesis <https://twiki.cern.ch/twiki/bin/view/Main/AtlasEdinburghGPUComputing>