GPU-Based Tracking Algorithms for the ATLAS High-Level Trigger

D Emeliyanov[1] and J Howard[2] on behalf of the ATLAS Collaboration

[1] STFC Rutherford Appleton Laboratory, Harwell Science and Innovation Campus, Didcot, UK

[2] University of Oxford, Oxford, UK

E-mail: dmitry.emeliyanov@stfc.ac.uk, j.howard1@physics.ox.ac.uk

Abstract. Results on the performance and viability of data-parallel algorithms on Graphics Processing Units (GPUs) in the ATLAS Level 2 trigger system are presented. We describe the existing trigger data preparation and track reconstruction algorithms, motivation for their optimization, GPU-parallelized versions of these algorithms, and a "client-server" solution for hybrid CPU/GPU event processing used for integration of the GPU-oriented algorithms into existing ATLAS trigger software. The resulting speed-up of event processing times obtained with high-luminosity simulated data is presented and discussed.

1. Introduction

ATLAS is one of the two general-purpose detectors in operation at the Large Hadron Collider (LHC) at CERN [1]. The detector is a layered design composed of a modular silicon and straw tube-based Inner Detector for tracking charged particles, solenoidal and toroidal magnet systems, a calorimeter for measuring energy deposits of particles, and a muon system for detecting and tracking muons.

The innermost part of the ATLAS detector is a silicon tracker, composed of a Pixel detector at the center and a silicon strip detector (SCT) surrounding it. Both the Pixel and SCT detectors are composed of thousands of silicon wafers tiled around the collision point. Each wafer contains an array of silicon sensors (two-dimensional pixels for the Pixel detector and one-dimensional strips for the SCT detector) which can pinpoint where charged particles have passed through and left charge deposits. Each SCT module consists of two layers, "axial" and "stereo", with the strips of opposite layers at a small relative angle to each other, allowing the tracking algorithms to determine where particles have passed through (with some ambiguity).

The ATLAS trigger system [2] performs the online event selection in three stages, referred to as Level 1, Level 2, and Event Filter. Level 2 and Event Filter are collectively referred to as the High Level Trigger. The Level 1 trigger is hardware-based and uses calorimeter and muon detector information to identify potentially interesting events and the so-called "Regions of Interest" (ROIs) within those events. ROIs are solid-angle regions of the detector with significant calorimetric or muon activity. They are used to seed the software-based Level 2 and Event Filter triggers, which have access to the Inner Detector tracking data and which apply further selection criteria to events based on data within the specified ROIs. The approximate rate reductions and latencies of each level are summarized in Table 1.

Table 1. Event processing rates at different trigger levels.

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

The Level 2 trigger is highly time-constrained by the incoming event rate and necessary output rate, and although event-level parallelization of the Level 2 trigger allows a slightly larger window for its operation, it is constrained by technological and economic factors. In the coming years, the LHC (and consequently the ATLAS detector) will see a large increase in instantaneous luminosity, up to $10^{35} cm^{-2} s^{-1}$ after the LHC Phase II upgrade, resulting in an increased event rate and tighter requirements on Level 2 run time. One possible avenue of optimization which is being explored is the use of General Purpose Computing on Graphics Processing Units (GPGPU Computing).

GPGPU computing is a relatively new technology which allows programmers to leverage the massively-parallel computing pipelines on modern graphics cards [3]. These cards have hundreds or thousands of cores and are well-suited to data-parallel algorithms. There are several implementations of GPGPU computing, all with similar data processing models, though for the purposes of this study we chose the NVIDIA Compute Unified Device Architecture (CUDA) toolkit due to its performance, support, and previous use in ATLAS. The typical model for GPGPU computing is a single "kernel" function written in a C dialect that is compiled for the GPU and run on an array of threads. These threads are organized into one-, two-, or three-dimensional abstractions (referred to as "blocks" in the CUDA SDK) in which each thread knows its position and can modify respective data. Data is copied from host system memory to on-GPU memory over the PCI Express bus and can be modified by threads running a kernel on the GPU before being copied back to the host memory. The primary limitations of GPU computing arise due to PCI Express bus bandwidth, branching in GPU code, and slow thread access to global device memory on the GPU, though these are becoming less important with advances in hardware and can be mitigated with programming techniques involving fast, block-local shared memory.

2. Level 2 Inner Detector Data Preparation and Tracking

A large portion of the Level 2 trigger's processing time is spent in data preparation and tracking. Data preparation consists of decoding hits in the silicon modules of the Inner Detector from a transport-optimized bytestream format, grouping these hits into clusters (which is necessary because particles often activate more than one silicon cell), and forming "spacepoints" from the clusters which represent points in three-dimensional space which are used for track reconstruction. The tracking algorithms then combine these spacepoints into track candidates and reject those tracks with a high statistical divergence from their constituent points.

2.1. Inner Detector Data Preparation in the ATLAS Level 2 Trigger

Data preparation at Level 2 is done on a per-ROI basis, which allows the trigger to request only the data from those silicon modules falling within an ROI, saving both time for processing and bandwidth for other systems using the detector readout. The data is read from a series of temporary storage systems referred to as Readout Buffers (ROBs). The data is encoded in a transport-optimized bytestream format and is subdivided into ROB fragments, which are segments of the bytestream coming from individual ROBs. Each fragment is further divided into datawords which are 32-bit for the pixel detector and 16-bit for the SCT detector. These words encode module identifiers (headers), hits, and module trailers. The format of each fragment is roughly summarized in Figure 1.

Module 1 Header
Module 1 Hit 1
...
Module 1 Hit M_1
Module 1 Trailer
...
Module N Header
Module N Hit 1
...
Module N Hit M_N
Module N Trailer

Figure 1. Bytestream format for a single ROB fragment [4].

It is important to note that the meaning of words within a fragment is context (position) dependent, and therefore any decoding algorithm must have access to at least some set of the neighboring words within a fragment.

The existing decoding algorithms are serial algorithms which run on the CPU and are primarily CPU-bound. The decoding algorithms for both the Pixel and SCT detectors are $\mathcal{O}(n)$, where n is the number of words in the bytestream. They iterate over ROB fragments, and then over the words within each fragment. While looping through an individual fragment the algorithm maintains a state containing information about which module it is decoding, assigning decoded hits to an appropriate in-memory container, and changing the decoding state whenever a module header or trailer is encountered. There is also error information that can be encoded in the bytestream, but these are fringe cases which are not considered here.

Once the hit words have been decoded and mapped to their appropriate modules, the clustering algorithm takes over, grouping adjacent hits within a module into clusters. The clustering algorithms for the Pixel and SCT detectors are similar in principle, but differ significantly in complexity and performance. The SCT clustering algorithm, because it operates in only one dimension, is $\mathcal{O}(n)$, where n is the number of hits per module. It consists of a single loop over the SCT strips, accumulating those which are adjacent and active into clusters. The pixel clustering algorithm, because it operates in two dimensions, is $\mathcal{O}(n^2)$. It consists of a double loop over each hit within a module, generating clusters for each hit, merging clusters when necessary due to a shared hit.

Finally, the Pixel and SCT clusters are converted to spacepoints by means of a simple geometric transform. The pixel clusters are rotated according to the module orientation and then offset by the module position. In the SCT case, clusters from each side of the module are combined into pairs and two-dimensional cluster positions are derived from these combinations, after which the translation procedure is identical to that of the pixels.

At each step in the data preparation process, some knowledge of the detector geometry is required, and hence any decoding, clustering, or spacepoint formation algorithm will require use of the ATLAS detector geometry. To solve this problem, a minimal version of the detector geometry was developed for the GPU which utilizes a hash table for efficient lookup of geometry information for the several thousand detector modules.

2.2. Track reconstruction in the ATLAS Level 2 Trigger

There are two tracking algorithms used in the ATLAS Level 2 Trigger: IdScan and SiTrack. For this study, SiTrack was chosen because it is based on a combinatoric approach and offers a more direct strategy for parallelization. The SiTrack algorithm includes the following steps [5]:

- Track seed formation
- Optional primary vertex reconstruction (not used in this study)
- Track seed extension and triplet formation
- Triplet merging into track candidates
- Clone track merging or removal

Track seeds are formed by generating pairs of space points from the two inner-most layers of the detector as shown in Figure 2. The pairs are then extrapolated using a straight line, and a cut is placed on the impact parameter with the beamline. Because charged particles curve in the magnetic field of the detector, this places a limit on the lowest transverse momentum, p_T, of the reconstructible tracks. To reduce the number of seeds, the primary vertex is optionally reconstructed by histogramming all track seed origins and finding the maxima. The seeds incompatible with any of the primary vertex positions are rejected. The remaining track seeds are then extended by finding space points in the outer layers which lie along the same trajectory (Figure 3).

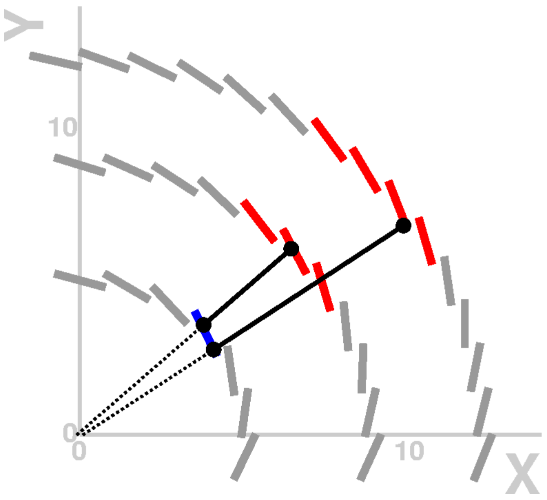

Figure 2. Track seed formation [2].

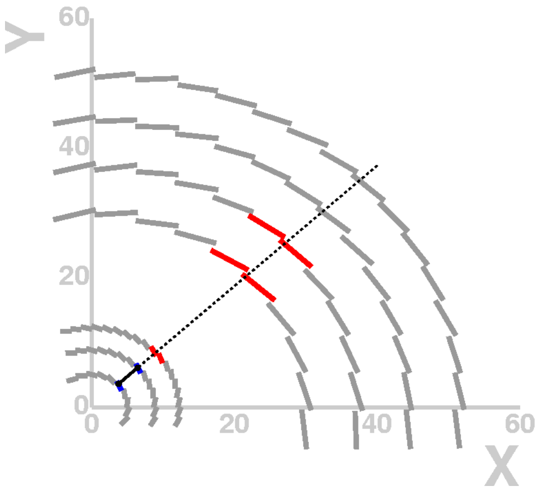

Figure 3. Track seed extension [2].

The original two-spacepoint seed and each spacepoint added at the seed extension stage form a unique spacepoint triplet so that a triplet "tree" is formed for each seed. All triplets which are generated for a given seed are then merged into track candidates. Finally, track candidates which share a large number of spacepoints are filtered by either the track with the largest number of space points or the smallest χ^2.

3. Implementation of GPU-accelerated algorithms

The GPU implementations of the data preparation and tracking algorithms differed significantly in difficulty based on the inherent potential for parallelism (or lack thereof) in the existing serial algorithms.

3.1. GPU-accelerated Data Preparation algorithms

The GPU version of the decoding algorithms is very similar to the CPU version, except that fragments from the bytestream are assigned to separate thread blocks and each word within the fragment is assigned to a separate thread, allowing all words to be decoded in parallel. Because the threads within a given block can access the same shared memory, all words for a given fragment can be copied into shared memory, allowing fast access to all words for threads trying to determine their context in a fragment. A global output buffer is created so that the hits for a module can be written to a location that can be accessed by the clustering routine. Atomic integer markers are used to ensure that module hits are only recorded once to global memory, reducing write access and decoding time. The complexity of the decoding algorithm is $\mathcal{O}(\frac{n}{m})$ where n is the number of words in the bytestream and m is the number of threads on the device. On modern GPUs with hundreds or thousands of cores, this denominator is significant enough to make a practical difference in decoding time.

The GPU version of the clustering kernel is run on the output of the decoding kernel. The GPU clustering algorithm is a cellular automaton similar to the algorithm used to cluster energy deposits in the LHCb calorimeter [6]. At the beginning of the algorithm, each hit is 'tagged' with an arbitrary but unique index (in this case the thread index). Each thread then iterates over all other hits, re-tagging the other hits if they are adjacent and have a smaller tag, and stopping its execution if another hit is adjacent and has a larger tag. This assures that there is at most one thread responsible for computing a given cluster. Once the state of the automaton no longer evolves, clustering is complete. This procedure is detailed in Figure 4. The complexity of this algorithm is $\mathcal{O}(\frac{nL}{m})$, where n is the number of active pixels, L is the average cluster size, and m is the number of threads on the device.

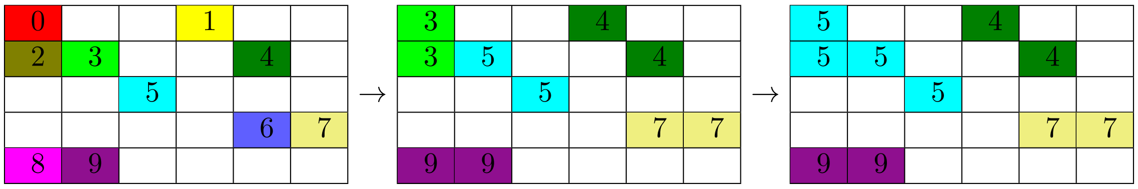

Figure 4. Evolution of cellular automaton clustering for a two-dimensional pixel array.

3.2. GPU-accelerated SiTrack algorithm

Due to the intrinsic parallel nature of the SiTrack algorithm, porting most of its steps to GPUs is relatively straightforward. From the very beginning of the algorithm workflow, combinining spacepoints results in data-parallel track seeds, whose further processing can be done independently.

The GPU-based seed formation is illustrated in Figure 5. For a given pair of detector layers (L_1, L_2) there is a 16×16 block of threads which perform seed generation. Spacepoints from each layer are divided into 16 "tiles" and a thread (i, j) combines spacepoints contained in the i-th tile from the layer L_1 with spacepoints in the j-th tile from the layer L_2. Seeds formed by the threads are stored in the common block-local buffer which is subsequently copied into an array in the global device memory.

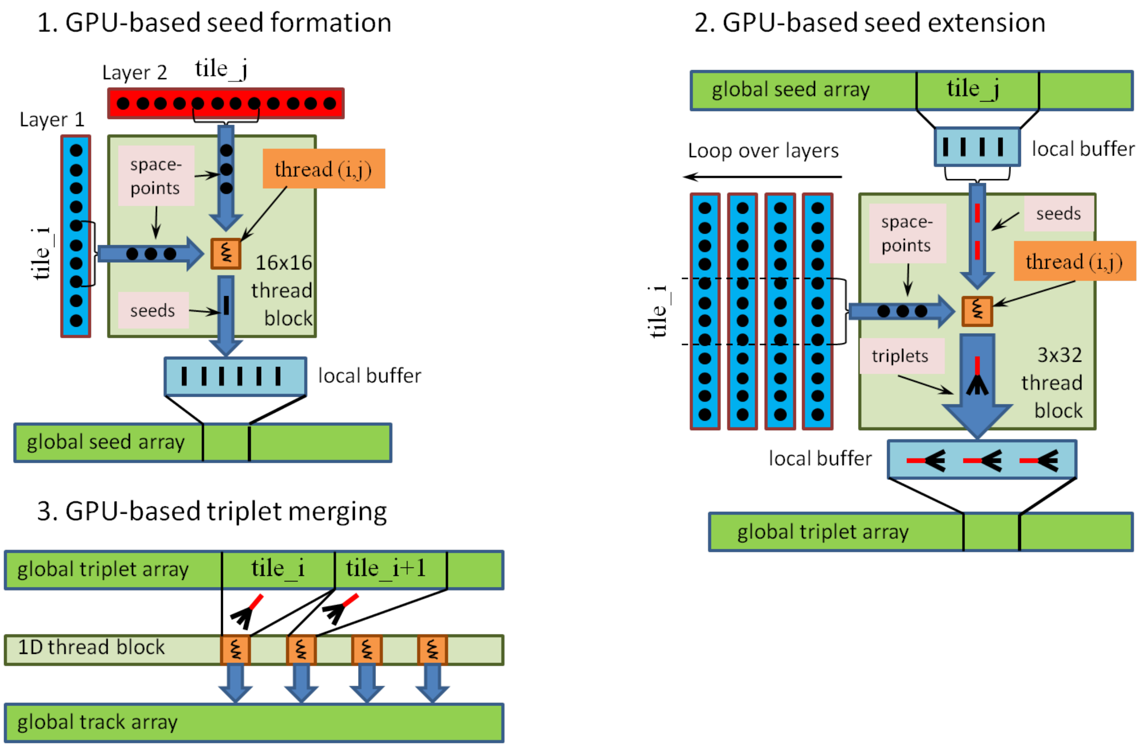

Figure 5. The design of the GPU-accelerated tracking algorithm.

The GPU-based seed extension and triplet formation also employ tile-based parallelization. As can be seen from Figure 5, 3×32 blocks of threads match tiles of 32 seeds with tiles of 3 spacepoints so that each thread handles a unique "seed-spacepoints" combination.

Finally, one-dimensional blocks of threads are used to merge triplets and create track candidates. At this stage, each thread handles a few triplet trees.

The last step of the SiTrack algorithm, clone track merging and removal, requires formation of pair-wise track candidate combinations which are then globally sorted in accordance with some track similarity measure. Due to strong data-dependence, this step is difficult to parallelize and, in this study, it was executed serially on a CPU.

3.3. Client-server architecture for hybrid CPU/GPU event processing

The implementation of GPU-based algorithms in the ATLAS Level 2 trigger requires careful consideration to integrate with existing trigger software while maintaining performance.

The ATLAS event processing and trigger software use a custom framework called "Athena", which is based on the "GAUDI" framework, originally developed for LHCb [7]. The Athena framework allows the trigger system to be emulated on existing data or Monte Carlo samples. However, the Athena framework has several limitations, which include a complex build system, high memory usage, and difficult threading. To avoid the complexities of integrating CUDA with Athena and to allow multiple instances of Athena to interact with a single GPU, a special "client-server" system was developed for this study.

The "client-server" solution illustrated in Figure 6 includes a multi-process server which executes GPU kernels and a few client processes which can request data be processed on the GPU using a specific algorithm. This solution allows Athena-based applications to utilize the GPU without adding any extra library dependencies since all client-server calls are made through a standardized POSIX interface. To maximize performance, the server runs on the same machine

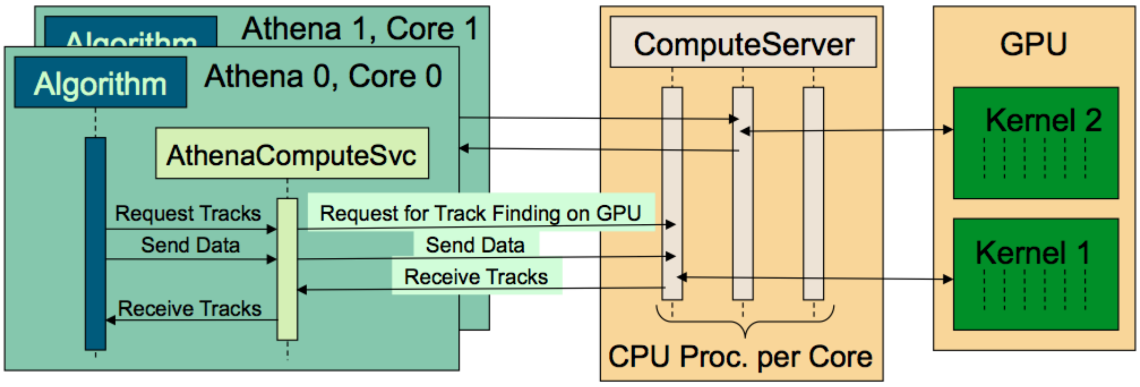

Figure 6. The "client-server" GPU/CPU processing model.

as the client processes and the memory shared between server and client processes is used for fast inter-process data transfer.

4. Results and Discussion

The tests of the GPU-accelerated algorithms and the "client-server" framework were conducted on a machine running Scientific Linux 5, Athena 17.1.0, and CUDA 4.0. The machine specifications are listed in Table 2.

Table 2. Test machine specifications.

Component	Specification
CPU	Dual Quad-Core Intel Xeon E5620 @ 2.40 GHz
Memory	24 GB
GPU	NVIDIA Tesla C2050 @ 1.3 GHz, 3 GB RAM

The data used for testing were high-luminosity Monte Carlo samples which simulate post-upgrade detector conditions.

As can be seen from Figure 7, the GPU-based processing results in an approximately 26x decrease in data preparation processing time for high-luminosity $t\bar{t}$ samples. As expected from the complexity analysis, the GPU processing time has very little dependence on input data volume. This is a common feature of GPU-based algorithms, and the processing time is limited only by the available bandwidth of the PCI Express bus and the multiplicity of processing cores on the GPU, both of which improve "for free" with new hardware once the algorithm has been parallelized.

Track reconstruction timing also shows improvement with respect to the CPU. Figure 8 shows a modest 2-3x speedup when using the GPU-based parallel algorithms. The computation time still has a large dependency on data input volume because the clone track merging/removal step of SiTrack is still performed on the CPU in all cases. These parts of the SiTrack algorithm are not dominant contributors to processing time when data preparation and tracking are run on the CPU, but when running the parallelized algorithms on the GPU, these steps take several times longer to run than the entirety of the GPU-parallelized portion. This behavior is a rather straightforward demonstration of the performance limits imposed by Amdahl's Law.

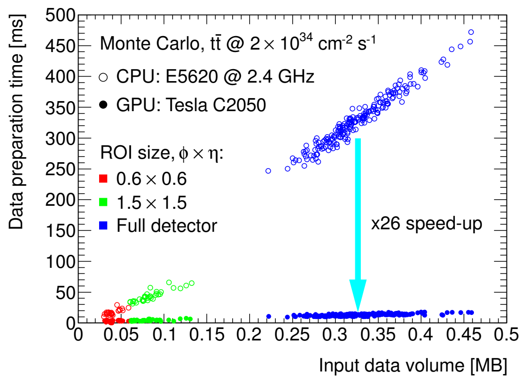

Figure 7. Performance improvement for data preparation steps.

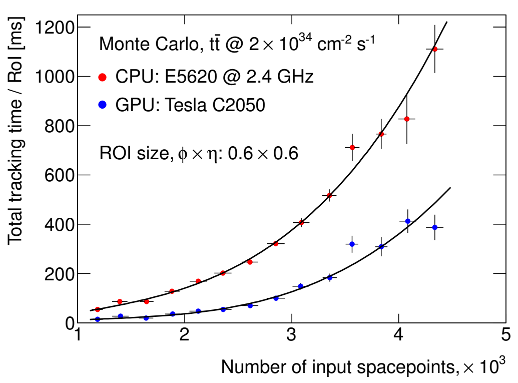

Figure 8. Performance improvement for tracking steps.

The total performance benefit of the data preparation and tracking steps used in conjunction with the "client-server" framework is demonstrated in Figure 9. It shows a nearly linear increase in event throughput as a function of trigger job multiplicity, but does demonstrate a saturation effect, most likely due to the limited number of streaming multiprocessors on the GPU. This does imply a limitation to the client-server model, though the more pressing constraint in practice is the number of trigger jobs which can be run on each machine, as they put high demand on system memory.

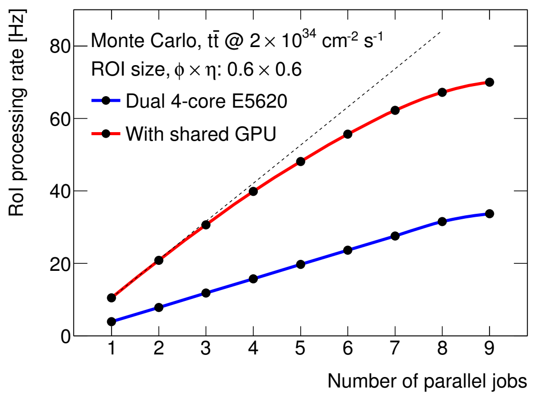

Figure 9. Total processing throughput as a function of trigger job multiplicity for the "tauNoCut" trigger menu item.

5. Conclusion

We have demonstrated the viability and potential performance benefits of using GPU-based algorithms for data preparation and tracking in the ATLAS Level 2 trigger. We have presented parallelization strategies for existing serial algorithms and a strategy for implementing these algorithms into the existing trigger framework. Tests with high-luminosity simulated data have shown increased performance in data preparation and tracking, and demonstrated the viability of the parallelization and implementation strategies.

Future work will focus on porting the implementation to OpenCL to allow testing on a wider variety of hardware, including ATI FirePro cards and many-core CPUs.

References

[1] ATLAS Collaboration, The ATLAS experiment at the CERN Large Hadron Collider. JINST 3 (2008) S08003.

[2] ATLAS Collaboration, Expected Performance of the ATLAS Experiment: Detector, Trigger, and Physics, 2009.

[3] I. Buck *et al.*, Scalable Parallel Programming with CUDA, Queue 6, 2 (March 2008), 40-53.

[4] C. Bee, et al., The raw event format in the ATLAS Trigger and DAQ. Technical Report ATL-DAQ-98-129, CERN, Geneva, Oct 1998.

[5] S. Armstrong, *et al.*, Algorithms for the ATLAS High-Level Trigger, IEEE Trans. Nucl. Sci., Vol. 51, No. 3, 2004.

[6] V. Breton, *et al.*, A clustering algorithm for the LHCb electromagnetic calorimeter using a cellular automaton. Technical Report LHCb-2001-123, CERN, Geneva, Sep 2001.

[7] G.Barrand, *et al.*, GAUDI - A software architecture and framework for building LHCb data processing applications, Computer Physics Communications 140 (2001) 45-55.