FATSEA – An Architectural Simulator for General Purpose Computing on GPUs

K.E. Østby[1], J.L. Aragón[1], J.M. García[1], and M. Ujaldón[2]

[1] Computer Engineering and Technology Dept.
University of Murcia (Spain)

[2] Computer Architecture Dept.
University of Malaga (Spain)

Abstract. We present FATSEA, a functional and performance evaluation simulator written in C++ to handle kernels written in the CUDA programming language aimed for GPGPU computing. FATSEA takes a Parallel Thread eXecution (*PTX*) code as input, which is a device independent code format generated by the Nvidia CUDA compiler, to validate results and estimate performance on Nvidia platforms. This paper shows results on a G80-based architecture for a set of well-known kernels to illustrate the usefulness of our framework while performing a preliminary validation for the tool.

1 Introduction

Many-core Graphics Processing Units (GPUs) constitute nowadays a solid competitor for multi-core CPUs on the road towards high-performance computing on a tight budget. Furthermore, with the addition of new programming environments such as CUDA [1] and OpenCL [2], the amount of general purpose applications that have been ported to GPUs is increasing dramatically. The understanding of GPUs has been thoroughly researched during this evolution, but with vendors poorly revealing the underlying microarchitecture, there is still a long way to walk.

FATSEA (Functional And Timing Simulator for Emerging Architectures) incarnates our contribution to shed a light in some of those dark areas. The first attempts to simulate GPU architectures were made in the context of GPUs considered as classical rendering devices, where Attila [3] and QSilver [4] constitute two encouraging examples. From here, the lack of a standardized simulation infrastructure, including compiler, profiler and benchmarks, is a key limiting factor for a better understanding of GPUs from a general-purpose perspective (GPGPU).

In parallel with our infrastructure, three other simulating tools have been developed as well. First, **Barra** is an interesting approach based upon the SystemC simulator framework Unisim [5] to provide a G80 functional simulator, but without accompanying a timing model. It directly executes the binary code, with the severe implications of not having available some of the unknown instructions. Second, **GPUSim** is a functional and timing GPU simulator based upon SimpleScalar [6] and is modeled to match the Nvidia 8600GTS architecture. Being closer to our approach, it includes the PTX front-end within the simulator itself, and therefore lacks of a certain generality and modularity which favours a fast adaptation to future architectural developments. Finally, **Ocelot** [7] is an open source infrastructure developed for understanding data parallel GPU applications which has recently been upgraded to enable PTX emulation [8]. However, since the main focus of Ocelot is workload characterizations, the developers have naturally omitted architectural simulations into their infrastructure. In that respect, Ocelot is more related to virtual machines than computer architecture simulators. The mechanics of Ocelot is to use the LLVM [9] infrastructure in order to compile PTX into machine code, and executes one block at the time. Thus, to that manner, FATSEA is closer to Ocelot for being (a) vendor agnostic, and (b) easily adaptive to future developments and novel ideas, however, the principal usage of FATSEA is to work as an architectural simulator such as GPUSim which allows for research on GPU architectural optimizations. Finally, by introducing the separation between the functional and timing model, we may also disable the timing model to work as a purely functional simulator.

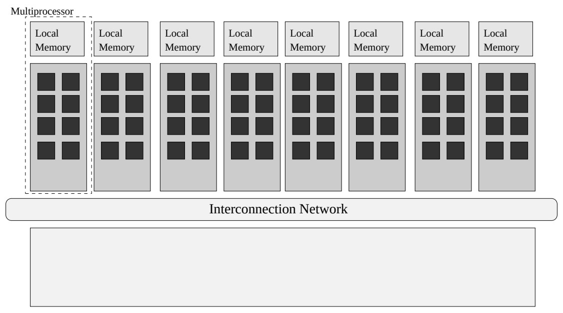
2 The Tool : FATSEA

The division of labor between the functional and timing model in FATSEA is performed by a driver which acts as an arbiter for the execution, keeping track of which events are to occur at a certain cycle. These events are either a memory request or a memory respond. With the memory transactions kept in a priority queue, the driver code can schedule the functional part of the simulator to execute the delta cycles between current time and the incoming event. This way, the functional side only needs to keep track of a tuple consisting of the current cycle and the threads in blocked state, in order to ensure timing accurate execution.

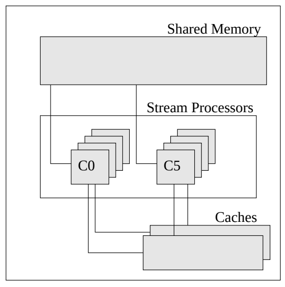
The functional model yielding to the timing model will happen if one out of two conditions occurs. (1) When it has executed the scheduled amount of cycles assigned by the driver code, or (2) when the functional module encounters an instruction which produces a side effect into the system, causing the threads to evolve to a blocking state. Instructions affecting the timing model such as load/stores are left to the latency and bandwidth features of a specific memory system, which can this way be replicated with high fidelity.

On an architectural level, the simulator is currently configured to match a G80-like architecture, where the GPU consists of a variable amount of multiprocessors, connected to an interconnection network holding data requests as seen in figure 1. The memory hierarchy comprises a shared memory local to each multiprocessor, and a global device memory including a cache per multiprocessor. The current version of the simulator assumes that the amount of threads in-flight will be enough to hide memory latencies, and since there is no memory coherence control in modern GPUs, this issue is also ignored by the simulator.

The execution model of FATSEA is inspired by CUDA and OpenCL, where the concept of blocks and threads are borrowed as a way of scheduling the workload amongst the available resources. When a new kernel is scheduled to be



(a) Overall System



(b) A single multiprocessor

Fig. 1. A high-level description of the underlying simulated architecture by FATSEA.

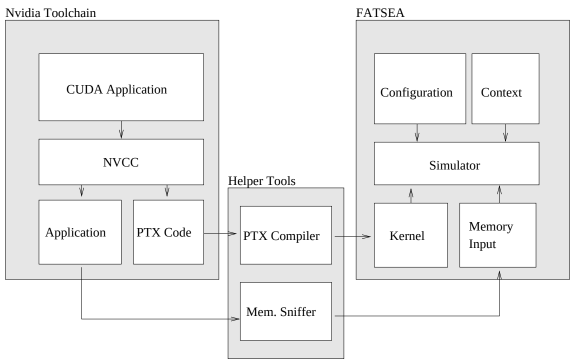


Fig. 2. Auxiliary tools developed and its relationship with CUDA modules.

executed, it does so by first uploading the kernel to main memory, and updating the kernel to address table, keeping track of the location of a kernel with a given ID in main memory. Afterwards, a set of *Blocks* are inserted into the global block queue. The blocks are data structures consisting of the following tuple *(blockid(x,y), kernel ID, number of threads (x,y,z), required resources)*. Then, a block scheduler attempts to schedule the blocks onto the available multiprocessors. When doing so, the block scheduler takes into account the required resources that the block needs to execute, such as registers, hardware barriers, and threads. If it finds a multiprocessor with the required amount of resources available, it dispatches the block onto that multiprocessor, where it is expanded as a set of lightweight threads, the basic work unit.

To experiment using the simulator framework, several auxiliary tools have been developed as well (see Figure 2). To start a new project, the CUDA code has to be written, and parsed through the CUDA *nvcc* compiler, which generates the files needed by our framework, namely the PTX code and the CUDA executable file. The PTX code is then compiled into an binary instruction format developed for FATSEA to ensure platform independency. A memory sniffer is implemented as a library which gets loaded in-front of the CUDA library *libcudart*. The configuration file contains the values for settings in the simulator, and

Feature	Value
Local Memory Size	16 Kb
Global Memory Size	500 Mb
L1 Cache Set Associativity	4
L1 Cache Replacement Policy	LRU
L1 Hit Latency	4 cycles
Local Memory Latency	4 cycles
Device Memory Latency	300 cycles
Cores Per Multiprocessor	8 cores
Thread Scheduling Algorithm	R. Robin

Table 1. Parameters describing our target GPU architecture.

Name	Block Dim.	Grid Dim.	Total
MersenneT	128x1	32x1	3968
BlackScholes	128x1	480x1	61440
FFT8	64x1	1-200x1	61-12800
MM	4x4-22x22	1x1	4-484
Sgemmn	16x4	4x1-30x7	256-13440
dwtHaar1D	1-512x1	2-512x1	2-262144

Table 2. PTX Kernel Properties.

the context file controls the execution environment (grid and block size, memory input files, and more).

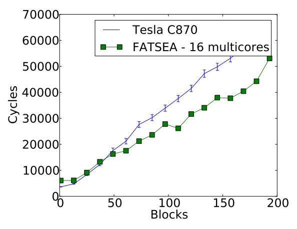
3 Experimental Results: Validation

The benchmarks used to validate the GPU simulator and the compiler consists of the kernels found in table 2. The Fast Fourier Transform (FFT) developed by Volkov *et al.* [10] is included due to being a computationally bound kernel. The Matrix Multiplication (MM) kernel is included in order to investigate the correctness of a single multiprocessor. Furthermore, the Sgemmn [10] kernel is an optimized matrix multiplication kernel, included due to its use of local memory and barriers. Finally, Mersenne Twister (MersenneT), BlackScholes [11] and dwtHaar1D are all benchmarks from the Nvidia CUDA SDK, and thus representative candidates for calculations on GPUs.

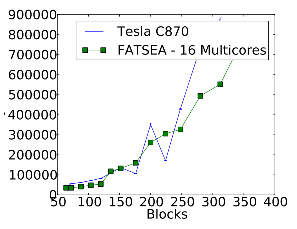
The configured architecture used to run the included benchmarks can be found in figure 1 and table 1. The thread execution model of FATSEA is currently utilizing a model based upon scoreboards [12], where threads are not placed in a blocked state until they try to operate on the data requested in a previous memory transfer. The architecture specified is configured to be similar to the newer architectures of Nvidia, and the results in figure 3 are compared to the Tesla C870.

Characteristic of the simulator is that it tends to underestimate the execution time when the working set is small, or when the benchmarks are not very dependant on accessing global memory, a phenomenon which is observed in the BlackScholes benchmark (fig. 3(c)). As can be seen in the Sgemmn (fig. 3(b)) and the MM (fig. 3(d)) benchmarks, when the kernel is utilizing the local memory or avoids a lot of global memory accesses, the simulator has a tendency to underestimate the cycle count. Further observable, an uneven working set produces spikes, as can be seen in the BlackScholes kernel when a variable number of options are computed. Spikes are generated by kernels doing some initial work and competing for resources before some of the threads gets terminated without doing any productive work. This happens in the cases where the problem set is not suitable for the kernel, nor the architecture.

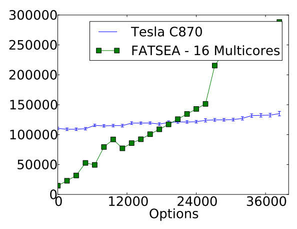
Seeing how FATSEA underestimates kernels with small working sets, there are unknown variables in the startup phase of the simulation as well. This might be due to inaccuracies in the instruction pipeline, since the simulator assumes an ideal instruction pipeline to avoid hazards that could be introduced by the PTX code, but that could be further optimized by the Nvidia PTX assembler. Moreover, as the GPU can avoid pipeline hazards by scheduling the threads in a round robin fashion, it is an assumption that will work fine whenever the GPU has sufficient amount of workload. Volkov *et al.* [10] show that the instruction latency in modern GPUs is about 20 cycles, and thus, in order to hide the pipeline latency, each multiprocessor only needs 160 different non-blocked threads to be able to totally hide the pipeline latency in the worst-case scenario. Moreover, since the simulator tends to overestimate the cycle count where the working set is large and the kernel involves a lot of memory transfers to global memory, it suggests that there are some unknown mechanisms such as a prefetching mechanism with regards to global memory.



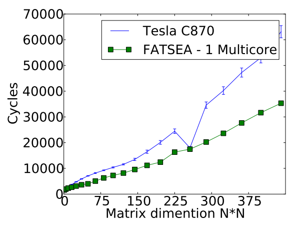
(a) FFT



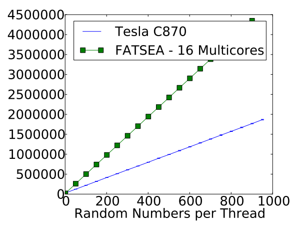
(b) SGEMMN



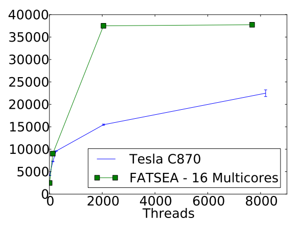
(c) BlackScholes



(d) MM



(e) Mersenne Twister



(f) dwtHaar1D

Fig. 3. Comparing the behaviour of six kernels on a Tesla machine (upper purple) and FATSEA (lower green). Number of cycles are represented in the vertical axis against the problem size in the horizontal axis.

4 Conclusions and future work

In this paper, we have presented a new framework for research on GPGPU devices from an architectural viewpoint, which opens the door for further research on architectural optimizations. Using a small set of benchmarks, we have implemented a well-known GPU architecture according to the freely available documentation from both industry and academia. We have also demonstrated that some platform details are still unknown under the hood: the simulated model is accurate when it comes to benchmarks requiring little memory overhead, but far from optimal when memory transfers predominate. Our future work will handle the interconnection network, thread scheduling and memory technologies for endowing FATSEA with a convergence road towards the behaviour of a real system.

Another direction points to novel mechanisms to further improve the architecture. For example, one of the pressing matters in modern architectures is the programming pattern that the lack of global synchronization mechanisms enforces. Although it is possible now with the current architectures of Nvidia to provide a basic global locking mechanism with the atomic instructions available to the developer, there is no native support for that. Furthermore, a way of solving data dependencies between blocks would be of great interest, currently solved by storing to global memory, and then terminating the kernel.

Overall, the challenge arises from the fact that the proposed solution would have to scale from few multiprocessors to a myriad without imposing too many changes neither on the current programming model nor the current architecture.

References

1. nVidia: NVIDIA CUDA Compute Unified Device Architecture . (2007) Rev. 1.0.
2. Khronos OpenCL Working Group: The OpenCL Specification, version 1.0.29. (8 December 2008)
3. Moya, V., Gonzalez, C *et al.*: Shader Performance Analysis on a Modern GPU Architecture. In: Proc. of the 38th Int. Symp. on Microarchitecture. (2005) 355–364
4. Sheaffer, J.W., Luebke, D., Skadron, K.: A Flexible Simulation Framework For Graphics Architectures. In: Proc. of the SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. (2004) 85–94
5. August, D., Chang, J. *et al.*: Unisim: An open simulation environment and library for complex architecture design and collaborative development. IEEE Comput. Archit. Lett. **6**(2) (2007) 45–48
6. Austin, T., Larson, E., Ernst, D.: Simplescalar: An infrastructure for computer system modeling. Computer **35**(2) (2002) 59–67
7. Kerr, A., DIamos, G., Yalamanchili, S.: Gpuocelot - A Binary Translator Framework for PTX. (October 2009) http://code.google.com/p/gpuocelot.
8. Kerr, A., DIamos, G., Yalamanchili, S.: A Characterization and Analysis of PTX Kernels. In: Proc. of Intl. Symposium on Workload Characterization. (Oct. 2009)
9. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois. (Dec. 2002)
10. Volkov, V., Demmel, J.W.: Benchmarking GPUs To Tune Dense Linear Algebra. In: Proc. of the 2008 ACM/IEEE Int. Conference on Supercomputing. (2008) 1–11
11. Podlozhnyuk, V.: Black Scholes Option Pricing, ver. 1.0. Nvidia SDK. (Apr. 2007)
12. Hennessy, J., Patterson, D.: Computer Architecture - A Quantitative Approach. Morgan Kaufmann (2003)