Technical Report

Calculating an upper bound on the finishing time of a group of threads executing on a GPU: A preliminary case study

Gurulingesh Raravi

Björn Andersson

HURRAY-TR-100701

Version:

Date: 07-11-2010

Calculating an upper bound on the finishing time of a group of threads executing on a GPU: A preliminary case study

Gurulingesh Raravi, Björn Andersson

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

Graphics processor units (GPUs) today can beused for computations that go beyond graphics and such usecan attain a performance that is orders of magnitude greaterthan a normal processor. The software executing on a graphicsprocessor is composed of a set of (often thousands of) threadswhich operate on different parts of the data and therebyjointly compute a result which is delivered to another threadexecuting on the main processor. Hence the response time ofa thread executing on the main processor is dependent on thefinishing time of the execution of threads executing on the GPU.Therefore, we present a simple method for calculating an upperbound on the finishing time of threads executing on a GPU, inparticular NVIDIA Fermi. Developing such a method is nontrivialbecause threads executing on a GPU share hardwareresources at very fine granularity.

Calculating an upper bound on the finishing time of a group of threads executing on a GPU: A preliminary case study

Gurulingesh Raravi and Björn Andersson
CISTER-ISEP Research Center
Polytechnic Institute of Porto
4200-072 Porto, Portugal
ghri@isep.ipp.pt, bandersson@dei.isep.ipp.pt

***Abstract*—Graphics processor units (GPUs) today can be used for computations that go beyond graphics and such use can attain a performance that is orders of magnitude greater than a normal processor. The software executing on a graphics processor is composed of a set of (often thousands of) threads which operate on different parts of the data and thereby jointly compute a result which is delivered to another thread executing on the main processor. Hence the response time of a thread executing on the main processor is dependent on the finishing time of the execution of threads executing on the GPU. Therefore, we present a simple method for calculating an upper bound on the finishing time of threads executing on a GPU, in particular NVIDIA Fermi. Developing such a method is non-trivial because threads executing on a GPU share hardware resources at very fine granularity.**

I. Introduction

Graphics processors were originally used only for graphics but they have evolved significantly during the recent decade as witnessed by the following. First, graphics processors double their performance every 6 months [1, page 1]; this should be compared with normal CPUs which double their performance every 18 months [1, page 1]. Consequently, graphics processors today offer significantly higher peak performance than a normal CPU. The most recent graphics processor, NVIDIA Fermi [2], has a peak computing performance of approximately one Teraflop [3]; this is approximately thousand times greater than a normal single-core processor in a normal PC. Second, graphics processors are able to perform general-purpose computations, programmed using CUDA APIs [1] with C; hence enabling "normal software developers" to use graphics processors for data-parallel programs. Therefore, today this type of processor can be thought of as a multicore processor; it has come to be called General Purpose Graphics Processor Unit (GPGPU) or simply GPU (the phenomenon is called GPU computing [2]).

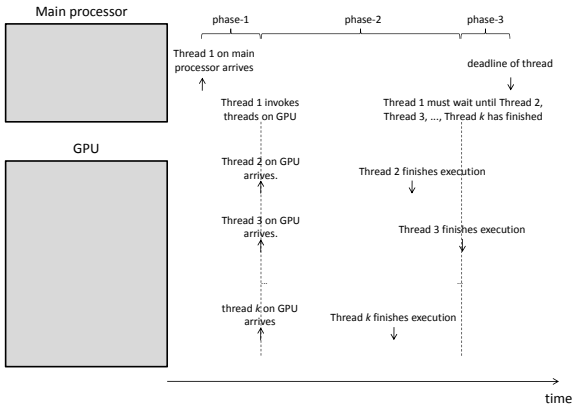
So far, the GPU has been marketed as a "supercomputer-at-your-desktop" but we believe that its use will also spread to embedded computer systems. A GPU however has no I/O capability and was not designed to run a normal operating system and therefore, a GPU is used as a co-processor to a CPU – the term CPU/GPU computing signifies this.

Figure 1(a) shows an example of the use of a GPU as a co-processor. A thread on the main processor arrives and performs some computations (for example reading sensors) and copies data from main processor's memory to GPU's memory. Then the thread on the main processor invokes the threads on GPU and suspends itself. The threads on the GPU execute in parallel on different data that they have been assigned and when these threads finish their execution, the thread on the main processor resumes execution. It copies the data from GPU's memory to main processor's memory and then uses this result (for example for actuation).

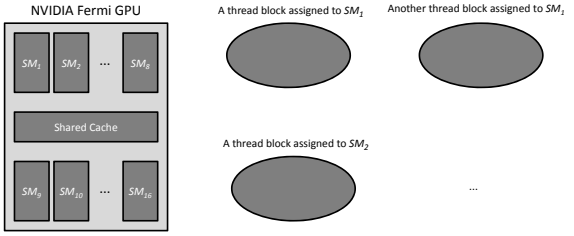
We can see that in order for CPU/GPU computing to be possible for hard real-time applications, three research problems must be solved:

P1. A method must exist for synchronizing the thread on the main processor and the threads on the GPU and the schedulability analysis on the main processor must take this synchronization mechanism into account. One could either (i) assign sub deadlines to the three different phases (shown in Figure 1(a)) or (ii) let the thread on the main processor suspend when not all threads serving it on the GPU has finished. The former approach transforms the problem to many scheduling problems with constrained-deadline sporadic tasks. For the latter approach, we can use scheduling theory which assumes that tasks can self-suspend [4], [5] for a time which is unknown but is upper bounded. A discussion on different models for describing such suspension in the context of GPU computing is available in [6].

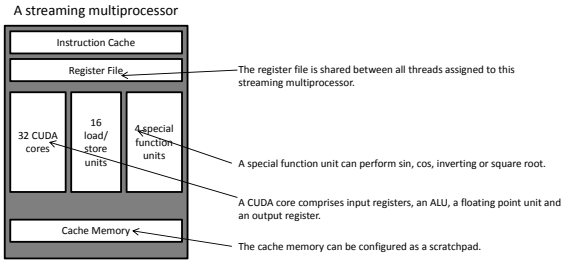
P2. A method must exist for determining if the GPU should be used to assist a thread on the main processor. This amounts to the task-assignment problem for heterogeneous multiprocessors which is known to be a very challenging problem (it is NP-hard and the standard use of normal bin-packing heuristics, such as first-fit, can cause poor performance [7]). But fortunately, in many practical scenarios, there are only two types of processors available (processor cores of the main



(a) A common use of a GPU. A thread executing on a main processor arrives and after some of its execution, it suspends and invokes multiple threads on the GPU. When these threads have finished, the thread on the main processor resumes execution again.



(b) The internals of NVIDIA Fermi GPU. It consists of 16 streaming multiprocessors which share a cache memory. To the right is shown thread blocks; a thread block is assigned to a streaming multiprocessor.



(c) A detailed view of a streaming multiprocessor.

Figure 1. The use of a GPU and its internals.

processor and the processor cores in the GPU) and for such situations, an efficient algorithm can be created [7].

P3. A method must exist for determining the finishing time of the group of threads executing on the GPU.

P1 and P2 have partly been addressed in previous research. The current research literature offers no method for P3 however and therefore, we will discuss P3.

Figure 1(b) shows the internals of a GPU; we consider the most recent one – NVIDIA Fermi. It comprises 16 so-called *streaming multiprocessors* (SMs) and a shared cache memory. Software threads are organized into so-called *thread blocks*, where a thread block is assigned to a streaming multiprocessor. A streaming multiprocessor may be assigned many thread blocks but a thread block cannot be assigned to two or more streaming multiprocessors. In order to solve P3 we therefore need to address two subproblems:

P31. Given that the assignment of thread blocks to streaming multiprocessors is known, compute, for each streaming multiprocessor, an upper bound on the finishing time of the threads assigned to this streaming multiprocessor.

P32. Assuming that the exact assignment of thread blocks is not known but some knowledge of the assignment heuristic is available (for example assign thread blocks in a round-robin fashion), compute, for each streaming multiprocessor, an upper bound on the finishing time of the threads assigned to this streaming multiprocessor.

Given that chip makers of GPUs currently do not publish the heuristic used for thread-block assignment, we focus on P31 and postpone P32 for future work. P31 cannot be solved through normal Worst-Case Execution Time (WCET) analysis [8] methods because they assume that all hardware resources of a processor is dedicated to a thread that executes. But this assumption is not true for a streaming multiprocessor. Figure 1(c) shows a detailed view of a streaming multiprocessor. A streaming multiprocessor comprises 32 Compute Unified Device Architecture (CUDA) cores; each CUDA core is composed of an ALU, a floating point unit, input registers and an output register. It is therefore possible for 32 threads to perform ALU operations in parallel on a single streaming multiprocessor. But there are only 16 load/store units; hence only 16 threads can perform load instructions in parallel. Analogously, only a limited number of trigonometrical computations can be performed in parallel. We can see that we need a method for WCET analysis which analyzes not only a single thread which has all hardware resources under its control but instead we must analyze a *set of threads* which *share* hardware resources.

Therefore, in this paper, we present a simple method for calculating an upper bound on the finishing time of threads assigned to a streaming multiprocessor. In order to take this first step, we limit ourselves to the study of a simplified version of a streaming multiprocessor in NVIDIA Fermi.

II. System Model

Program Structure: We consider the code structure shown in Figure 2 (derived from matrix multiplication code) for analyzing the finishing time of a set of threads. The '+' mark over the sub-block (involving LOAD, LOAD and a CUDA instruction) in Figure 2 indicates that the sub-block may repeat itself one or more times. The program structure of Figure 2 consists of:

1) an arithmetic instruction (say, initializing a register) that will be executed on a CUDA core followed by

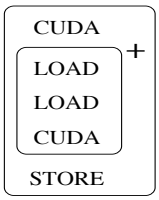


Figure 2. The template of the program's control flow. CUDA means an instruction that uses the CUDA processor (e.g., an ALU operation), LOAD is an instruction that fetches data from memory to a register and STORE is an instruction that stores the content of a register to memory.

2) a sub-block (which can occur one or more times) consisting of two memory accesses (say, reading two variables into registers) carried out by LOAD/STORE unit and an arithmetic instruction (say fused multiply and add) followed by
3) another memory access instruction (say, to store the result of fused multiply and add instruction).

Assumptions: In order to calculate an upper bound on the finishing time of a *kernel* (a set of thread blocks which serve a thread on the main processor), we make the following assumptions on the SM and its scheduling algorithm:

- Before run-time, tasks assigned to a SM are organized into groups of 16 threads (also known as a *warp*).
- Each instruction takes just one clock cycle to execute (including memory access instructions).
- There are no cache misses i.e., all the data that memory access instructions are interested in is found in cache (it is intuitive from the previous assumption).
- Each SM has 32 CUDA cores and 16 LOAD/STORE units i.e., in a clock cycle, each SM can either perform 32 arithmetic operations or 16 memory operations.
- At run-time, in each clock cycle, the scheduler of SM selects one or two warps for scheduling.
- Whenever there are warps available for execution, the run-time scheduler must select a warp (work-conserving). (Since we assume that each instruction takes just one clock cycle, every warp is available every time as long as its threads have not yet terminated.)
- As already mentioned, we assume that the control flow of the program is as specified by Figure 2; it comprises a CUDA instruction followed by rep_cnt sub-blocks and then a LOAD/STORE instruction. In addition, we assume that instructions are scheduled in a fair manner on sub-block level, that is, when threads compete for resources, an instruction belonging to sub-block k has priority over an instruction belonging to sub-block k+1 – this will be illustrated in Section III. (The assumption on fairness on sub-block level is probably not realistic for GPUs of today but this assumption has the benefit that it simplifies our analysis.)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Threads 1–16	C	L	L	C	S									
Threads 17–32	C			L		L	C	S						
Threads 33–48		C					L		L	C	S			
Threads 49–64		C								L		L	C	S
Clock Cycle														

a) One possible interleaving.

	1	2	3	4	5	6	7	8	9	10	11	12	13
Threads 1–16	C	L				L	C			S			
Threads 17–32	C		L				L	C			S		
Threads 33–48		C		L				L	C			S	
Threads 49–64		C			L				L	C			S
Clock Cycle													

b) Another possible interleaving.

Figure 3. Two possible interleavings/schedules when sub-block has appeared only once.

III. The new method

In this section, we present our method to determine an upper bound on the finishing time of n threads assigned to a single SM. We assume all these threads have the same program structure that is described in Section II. Recall that we make no assumption on the exact scheduling policy in a SM – we assume that it is work-conserving and sub-block level fair. Hence, for a given set of threads, there are many possible schedules (interleavings) that can be generated. To calculate an upper bound on the finishing time of a kernel, it is essential to know: "Given that a certain number of threads are assigned to a particular SM, what order of interleaving of these threads results in a maximum time (measured in clock cycles) to finish their execution". Figure 3 shows two such possible interleavings when 64 threads are assigned to a SM. Figure 3 shows, in each clock cycle, the instruction of different warps that is being executed by CUDA and LOAD/STORE units. We have used **C**, **L** and **S** to represent *CUDA*, *LOAD* and *STORE* instruction respectively. For example, in Figure 3(a), in the first clock cycle, CUDA instruction of thread 1-16 (say, warp1) and 17-32 (say, warp2) are being executed; in the second cycle, LOAD instruction of warp1 and CUDA instruction of threads 33-48 (say, warp3) and 49-64 (say, warp4) are being executed.

In the first case (Figure 3(a)), the scheduler is trying to finish the execution of one warp before switching to another warp but without violating its *work-conserving* property. In this case, the scheduler has scheduled the threads as follows: whenever possible, warp1 is executed, then warp2, then warp3 and when none of these warps can be executed, it executes warp4. In the second case (Figure 3(b)), the scheduler is trying to give a fair share of the resources to each warp by executing one instruction each from every warp and at the same time exploiting the parallelism whenever possible (to preserve the work-conserving property). As we can observe form this example, the first schedule has the

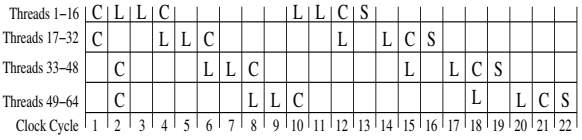


Figure 4. A schedule when sub-block is repeated twice.

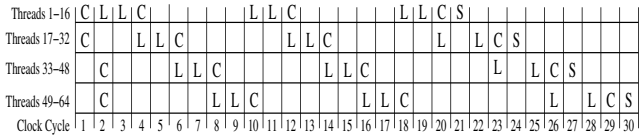


Figure 5. A schedule when sub-block is repeated thrice.

worst case finishing time as it takes 14 clock cycles whereas the second schedule takes only 13 clock cycles to finish.

Hence, we conjecture that the worst case interleaving happens for our program structure when the scheduler tries to schedule threads (group of 16 for the architecture under consideration) so as to finish the execution of a group of 16 threads in a particular order (as shown in Figure 3(a)).

Now, considering our conjectured worst possible interleaving for a given number of threads on a SM, we discuss the worst-case finishing time for a given number of threads on a SM. From the schedule shown in Figure 3, we can observe that, for the program structure under consideration, each warp (except the first one) needs 3 additional clock cycles to finish its execution compared to its previous warp. Note that in Figure 3, the inner sub-block (consisting of LOAD, LOAD and CUDA instructions) appears only once in the program structure. This can be generalized to n threads: The total number of clock cycles needed to finish the execution of n threads when the sub-block appears only once in the program structure is: $\lceil \frac{n}{16} \rceil \cdot 3 + 2$.

Now, consider the same number of threads (i.e., 48 threads) but with the sub-block repeated multiple times. To understand the finishing times in such a scenario, consider schedules for the cases when the sub-block has repeated twice and thrice in Figure 4 and 5. As we can observe, the total number of clock cycles (required to finish the execution of all the 48 threads) for each repetition of sub-block increases by 8. This can be generalized to n threads: The number of additional clock cycles needed to finish the execution of n threads when the sub-block has appeared rep_cnt of times is: $\lceil \frac{n}{16} \rceil \cdot (rep\_cnt - 1) \cdot 2$.

Hence, with the help of above two relations, we conjecture that the total number of clock cycles needed to finish the execution of n_i threads with rep_cnt appearances of the sub-block assigned to SM_i is:

$$FT_i = \left(\left\lceil \frac{n_i}{16} \right\rceil \cdot 3 \right) + 2 + \left\lceil \frac{n_i}{16} \right\rceil \cdot (rep\_cnt - 1) \cdot 2$$

This equation gives the finishing time of n_i threads assigned to SM_i. As described earlier, a GPU consists of many such SMs (say m) and hence the maximum finishing time of a *kernel* is: $FT_{kernel} = \max(FT_1, FT_2, \ldots FT_m)$.

IV. Conclusion and Future work

We presented a method for calculating an upper bound on the finishing time of threads executing on an NVIDIA Fermi GPU. We left the following problems open: (i) proving mathematically the correctness of our stated upper bounds on finishing times, (ii) relaxing the assumption of fairness on sub-block level, (iii) relaxing the hardware assumptions to allow cache misses and longer latency of floating-point operations, (iv) generalizing the method to other control flows and to user-specified number of LOAD/STORE units and CUDA cores and (v) validating the output of our method by comparing it with experimental runs on real hardware.

Acknowledgments

This work was partially supported by ARTISTDesign Network of Excellence on Embedded Systems Design, funded by the European Commission under FP7 with contract number ICT-NoE-214373 and the Portuguese Science and Technology Foundation (Fundação para Ciência e Tecnologia - FCT) and the Luso-American Development Foundation (FLAD).

References

[1] "NVIDIA CUDA Compute Unified Device Architecture Programming Guide, available at http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf."

[2] "NVIDIA's Next Generation CUDA-ATM Compute Architecture: Fermi, http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf."

[3] "http://techreport.com/articles.x/17670."

[4] K. Bletsas, "Worst-case and best-case timing analysis for real-time embedded systems with limited parallelism," Ph.D. dissertation, The University of York, 2007.

[5] P. Gai, L. Abeni, and G. C. Buttazzo, "Multiprocessor DSP scheduling in system-on-a-chip architectures," in *14th Euromicro Conference on Real-Time Systems (ECRTS 2002)*, Vienna, Austria, Jun. 2002, pp. 231–238.

[6] K. Lakshmanan, S. Kato, and R. Rajkumar, "Problems in scheduling self-suspending," in *RTSOPS 2010: 1st International Real-Time Scheduling Open Problems Seminar in conjunction with the 22th Euromicro Intl Conference on Real-Time Systems*, Brussels, Belgium, Jul. 2010.

[7] B. Andersson, G. Raravi, and K. Bletsas, "Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors," CISTER research unit, ISEP/IPP, Polytechnic Institute of Porto, Porto, Portugal, Tech. Rep. HURRAY-TR-100505, May 2010.

[8] A. C. Shaw, "Reasoning about time in higher-level language software," *IEEE Trans. Software Eng.*, vol. 15, no. 7, pp. 875–889, 1989.