Makespan computation for GPU threads running on a single streaming multiprocessor

Kostiantyn Berezovskyi

Konstantinos Bletsas

Björn Andersson

HURRAY-TR-120710

Version:

Date: 07-17-2012

Makespan computation for GPU threads running on a single streaming multiprocessor

Kostiantyn Berezovskyi, Konstantinos Bletsas, 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 processors were originally developed for rendering graphics but have recently evolved towards being an architecture for general-purpose computations. They are also expected to become important parts of embedded systems hardware -- not just for graphics. However, this necessitates the development of appropriate timing analysis techniques which would be required because techniques developed for CPU scheduling are not applicable. The reason is that we are not interested in how long it takes for any given GPU thread to complete, but rather how long it takes for all of them to complete. We therefore develop a simple method for finding an upper bound on the makespan of a group of GPU threads executing the same program and competing for the resources of a single streaming multiprocessor (whose architecture is based on NVIDIA Fermi, with some simplifying assunptions). We then build upon this method to formulate the derivation of the exact worst-case makespan (and corresponding schedule) as an optimization problem. Addressing the issue of tractability, we also present a technique for efficiently computing a safe estimate of the worst-case makespan with minimal pessimism, which may be used when finding an exact value would take too long.

Makespan computation for GPU threads running on a single streaming multiprocessor

Kostiantyn Berezovskyi and Konstantinos Bletsas
CISTER/ISEP Research Unit
Polytechnic Institute of Porto, Portugal
Email: {kosbe, ksbs}@isep.ipp.pt

Björn Andersson
Software Engineering Institute
Carnegie Mellon University, Pittsburgh, USA
Email: baandersson@sei.cmu.edu

Abstract

Graphics processors were originally developed for rendering graphics but have recently evolved towards being an architecture for general-purpose computations. They are also expected to become important parts of embedded systems hardware – not just for graphics. However, this necessitates the development of appropriate timing analysis techniques which would be required because techniques developed for CPU scheduling are not applicable. The reason is that we are not interested in how long it takes for any given GPU thread to complete, but rather how long it takes for all of them to complete. We therefore develop a simple method for finding an upper bound on the makespan of a group of GPU threads executing the same program and competing for the resources of a single streaming multiprocessor (whose architecture is based on NVIDIA Fermi, with some simplifying assumptions). We then build upon this method to formulate the derivation of the exact worst-case makespan (and corresponding schedule) as an optimization problem. Addressing the issue of tractability, we also present a technique for efficiently computing a safe estimate of the worst-case makespan with minimal pessimism, for use when finding an exact value would take too long.

1. Introduction

The stream processing computational paradigm was conceived so as to allow efficient processing for a particular type of parallel applications (with minimal data dependencies) while simultaneously simplifying the parallel hardware architecture. Given a set of data (a *stream*), a series of operations (*kernel function*) is applied to each element in the stream. This paradigm applies very nicely to graphics and was partially implemented in graphics processing units (GPUs) [16]. In other words, GPUs were designed to execute a large number of threads (in the order of thousands or more) so that their joint execution provides a result to a user. These devices were originally used only for graphics, but have evolved significantly during the recent years. Today, they are also capable of performing general-purpose computations (for desktop applications) and are hence called *general purpose graphics processing units* (GPGPUs).

GPGPUs can be expected to also become important building blocks in embedded systems. This expectation is reflected in the OpenCL initiative [13] which aims to provide the necessary framework, as infrastructure, to embedded system developers. Lots of embedded applications could benefit from GPGPU computing. For example, in [6] the authors work on speed-limit sign recognition system that should be part of driver support solutions in future automobiles. This service seems to perform complex (including massively parallel) computations using CPU and GPU in the background and only notifies the human user in special important situations. In [8] the authors dwell on the more general problem of real-time robust obstacle detection. Such scenarios impose real-time constraints (in the form of *deadlines*) for the timely completion of the multi-threaded GPU-based computation. The literature offers some models for performance analysis of general-purpose computation on graphics hardware [14], [19], [1], [9], but all of them consider the average case. However, in real-time system design, it is the worst-case that is of interest.

Many schedulability analysis techniques today assume that the execution time of one thread is unaffected by the execution of other threads and also that we are interested in the timing of each thread. For GPUs however, these assumptions are false because the execution of one thread on a processor (for example a CUDA core in a streaming multiprocessor of NVIDIA Fermi) may require hardware units (e.g. for loading/storing the data) shared with other processors in the streaming multiprocessor. As we mentioned above GPUs were not designed for running a single thread to provide a result, so we are interested not in the timing of an individual thread but of a group of threads. Worst-case execution time (WCET) analysis [20] calculates an upper bound on the execution time of a program when it executes as the only program in a computer system. Therefore, such analysis gives a result which is independent of the execution of other threads. As already mentioned above, this is even more unrealistic for GPUs than it is for CPUs. Moreover, even if accurate simulators for the GPU architecture were available (and internal scheduling policies were documented in every detail – which is currently not the case), guaranteeing the satisfaction of real-time constraints (meeting the deadlines) necessitates accounting for the worst case, which might never be observed during

. This work was supported by the REGAIN project, ref. FCOMP-01-0124-FEDER-020447, co-funded by National Funds through the FCT-MCTES (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme ‘Thematic Factors of Competitiveness’).

simulation. Therefore, timing analysis techniques are needed which can determine the time that a group of threads may take to complete, under this computational model. In this paper we compute the worst-case makespan (further on referred to as the makespan) – the longest possible time interval between the moment when the "earliest" thread starts execution, and the moment when the "latest" one finishes.

The real-time community is interested in the deployment of GPUs in real-time systems. Focusing on the fact that GPUs were originally I/O devices, the execution model in [12] improves the response times of high-priority tasks that should be run on GPUs. In [11], resource-sharing protocols for using direct rendering on GPUs are presented. In many cases, the GPU is used as a *co-processor* to which certain functions are offloaded for speed up – and this is the use we are most interested. Analytical approaches for such systems (as the one in [5]) typically rely on external analysis to derive worst-case execution times on the GPU. Our work does exactly that, hence it can complement such analytical approaches.

In the remainder of this paper, Section 2 presents the platform description and the problem formulation. Section 3 introduces advanced notation and terminology. Section 4 offers a new, fast but pessimistic method, for calculating an upper bound on the makespan for a single streaming multiprocessor. Section 5 formulates a binary Integer Linear Programming (ILP) optimization problem for finding the exact worst-case makespan. Section 6 presents the results of the experiments. Section 7 concludes and describes future work.

2. Architectural model and assumptions

We target a streaming multiprocessor which is based on the NVIDIA Fermi [15], a recent hardware architecture of GPUs, that are capable of general-purpose computations. Although we consider a single streaming multiprocessor in our analysis (while NVIDIA Fermi includes 16 of them), it already consists of a large number of components. In our simplified model, a streaming multiprocessor comprises:

- Multiple (C in count) programmable Compute Unified Device Architecture (CUDA) cores (each comprising an arithmetic logic unit, a floating point unit, input/output registers). CUDA cores of a given streaming multiprocessor perform computations in parallel with each other.
- Supporting units, that load and store data from/to cache or DRAM. These "load/store units" (L in count) allow source and destination address calculation for up to L threads per clock cycle per streaming multiprocessor.

Traditionally, the "entity" of computation is a thread, but it is important to emphasize that GPU threads differ greatly from CPU threads as the respective hardware architectures are drastically different. CPUs have branch prediction (so that a thread does not have to wait for the result of a branch), speculative execution (so as to perform computations before even being sure if the result will be needed), out-of-order execution (wherein an instruction can be performed as soon as its operands become available), substantial cache hierarchy (so as to read/write the data faster in the average case), prefetching (to get the data earlier). All these hardware optimizations, that CPUs are built around, aim to minimize the average latency. In contrast let us consider a GPU-thread which is running and needs to access the main memory. It takes hundreds of clock cycles to do that [17] and the GPUs do not have such a sophisticated architecture, like the one earlier described, that would help run a thread faster. Therefore, whenever the GPU thread sends a request to the main memory, the processor switches to executing another thread. In the general case, whenever any GPU thread stops for some reason, if there is enough work to do, we can always keep the streaming multiprocessor busy in the meantime. In this way, throughput is good, even if the processing of a single thread is not always fast. Instead of minimizing latency (like CPUs do), GPUs have a large number of computational units and switching between threads "hides" the latency and consequently increases the efficiency. Another important aspect is that GPU threads are much more "light-weight" than ordinary CPU threads, because context-switching between them does not involve updates to operating system data structures and takes very few clock cycles (vs hundreds for the CPUs). One of the reasons that context-switching between GPU-threads is fast is that all of them execute the same program (the "kernel") in parallel. This is also why, in GPU computing, it is much more convenient to think not in terms of individual threads but, instead, in terms of another entity of computation, the *warp* – a group of S threads, each of which executes the same kernel concurrently. A streaming multiprocessor processes warps, while its CUDA cores and load/store units process the corresponding threads. All threads of all warps, which are running on a given streaming multiprocessor, execute the same kernel [15].

Instruction latencies are largely dependent on the locality of the data. However, since NVIDIA Fermi has a relatively large amount of on-chip memory [8], we make the simplifying assumption that there exists no off-chip traffic (no cache misses). Regardless of the type of a particular kernel instruction (i.e. whether it is for a CUDA core or a load/store unit), we assume that it takes a single clock cycle to perform it and during this period of time, the corresponding computational unit deals with this instruction only (we will relax this assumption later).

As mentioned above, the resources of a streaming multiprocessor are shared between warps, but the actual scheduling policy is not published by the chip makers. It is reasonable, however, to assume that it is *work-conserving*. This means that whenever there are free CUDA cores or load/store units, they should be utilized by the available warps.

Let us summarize all the most important considerations and assumptions as follows:

- A streaming multiprocessor consists of two types of hardware units: load/store (L in count), and CUDA cores (C in count).
- Threads are organized into warps. All the threads (S in count) of the same warp execute in parallel.
- All threads from all warps execute the same kernel.
- There are no cache misses (so we do not have to estimate memory latency).
- Any instruction, takes no more than a single clock cycle. An instruction holds the resource exclusively (no other instruction can use it).

- The warps are scheduled in a work-conserving manner.

3. Notation

Early works [7], on using GPUs for general-purpose computation, contain a lot of reverse engineering efforts and in terms of programming, everything was developed by hand in assembly code. The positive aspect of the low-level coding was that the developers had better knowledge of how their programs would use the hardware units of the GPUs. Later, researchers began to use the OpenGL graphics interface [18] for general-purpose computation. This was also tedious because, although in most cases the code did something completely different, it still had to be written as if it were graphics computations.

Nowadays, the programming model for GPU computing is moving towards that of the high-level programming languages, but for our work, we still need good understanding of how the kernel (that is serving general-purpose applications) uses the computational units of the streaming multiprocessor. One option for the developer is the Parallel Thread Execution (PTX) – a low-level virtual machine and instruction set architecture, that is supported by NVIDIA Fermi [15], and was designed to provide developers with an interface for such programming languages as C++, OpenCL, DirectCompute, Fortran, C [15]. The compiler translates code written in these high-level languages into pseudo-assembly PTX-code, that is executed on a virtual machine; another compiler, that is the part of the graphics driver, translates the PTX-code into the target hardware instruction set – a binary code which can be run on the computational units of a streaming multiprocessor.

In this work, we introduce (and reason on the basis of) an abstraction of the PTX-code, which we term *kernel instruction string* – a sequence of "L" and "C" symbols, each of which represents a hardware instruction that should be performed on load/store unit ("L"-instruction) and CUDA core ("C"-instruction) respectively. For example, the kernel instruction string "LC" specifies that an instruction should be carried out by the load/store unit, followed by an instruction that should be performed on the CUDA core.

Since GPUs evolve rapidly, via PTX, NVIDIA provides a stable layer of pseudo-assembly language to developers, while remaining free to change the underlying instruction set later, if necessary. However, this is not the only aspect of GPU architecture that is subject to significant changes from generation to generation. The term *compute capability* specifies the level (designated by a number in the format $x.y$) of revision of the NVIDIA GPU core architecture. In particular, the number of computational units of each kind in a streaming multiprocessor may be different in devices with different compute capabilities. For example, for devices of compute capability 2.0 and 2.1, the number of CUDA cores in a single streaming multiprocessor is 32 and 48 respectively [17]. Since we aim to make our approach as general as possible, we next introduce the following variables, for the purposes of addressing streaming multiprocessors of different compute capabilities:

Let σ_L and σ_C denote the maximum number of warps that can simultaneously (i.e. during the same clock cycle) execute an "L"- or, respectively, a "C"-instruction on a streaming multiprocessor. Obviously, σ_L and σ_C depend on the number of load/store units (L) and the number of CUDA cores (C) in the streaming multiprocessor and also on the warp size (S):

$$\sigma_L = \frac{L}{S} \tag{1}$$

$$\sigma_C = \frac{C}{S} \tag{2}$$

Equations (1) and (2) assume that all threads in a warp execute the same instruction ("C" or "L") within the same clock cycle. However, if the number of computational units of some type is less than the warp size S, this is simply not possible. For example, a streaming multiprocessor of compute capability 2.0 has only $L = 16$ load/store units while the warp size is $S = 32$ (and, respectively, $\sigma_L = \frac{16}{32} = \frac{1}{2}$). Half the threads of the warp (a *half-warp* [17]) would execute an "L"-instruction in one clock cycle and the other half in another one. If these clock cycles are consecutive, this would be equivalent to the "L"-instruction taking two clock cycles to execute. To remove the difficulty arising from having fractional σ-values or, equivalently, having dissimilar latencies for "L"- and "C"-instructions (and also in order to simplify the construction of the ILP problem formulation, at a later stage), we introduce a transformation of the kernel instruction string as follows:

Assume that a single streaming multiprocessor includes U units of some kind. We know [15] that in NVIDIA Fermi and its ancestors (GT200, G80) the number U is a power of 2, $U \leq S$, and $S \, mod \, U = 0$. Hence corresponding $\sigma_U = \frac{U}{S} = \frac{1}{n}$, where $n \in \mathbb{N}$. Multiplying both sides of the equation by n, we get

$$n \, \sigma_U = 1$$

We can transform the instruction string for our kernel by replacing each "U"-instruction with n "U"s (each one corresponding to each "sub-warp" of $\frac{S}{n}$ threads) and additionally assuming that $\sigma_U = 1$ (the transformation is equivalent because n sub-warps of a warp, execute the "U"-instruction in mutual exclusion [2]). For our example of a streaming multiprocessor of compute capability 2.0, where $L = 16$, $C = 32$, $S = 32$, $\sigma_L = \frac{1}{2}$, $\sigma_C = 1$, the instruction string "LC" will be transformed to "LLC". (In other words, the original "L"-instruction is replaced by 2 consecutive "L"-instructions) and the value of σ_L will be changed to $\sigma_L = 1$ (Figure 1). We apply the same technique (at the cost of some pessimism) to the occasional CUDA instruction that takes more than one cycle.

$$\underset{\sigma_L = \frac{1}{2}}{LC} \quad \Rightarrow \quad \underset{\sigma_L = 1}{LLC}$$

Fig. 1. Transformation of the kernel instruction string

4. Pessimistic makespan derivation

This section introduces an approach with very low computational complexity for deriving an upper bound on the make-

Clock Cycle	1	2	3	4	5	6	7	8	9
Warp 1	L				L	C			
Warp 2		L				L	C		
Warp 3			L				L	C	
Warp 4				L				L	C

Fig. 2. Possible schedule ($\sigma_L = \sigma_C = 1$)

span of a group of threads executing on a streaming multiprocessor. This approach is pessimistic but its output may serve as input to other, less pessimistic, derivations (as later shown).

The pessimistic derivation formulated in this section is based on the fact that a streaming multiprocessor is used most inefficiently when, in a given clock cycle, all warps contend for the same type of computational unit. In that scenario, the computational units of other types are “wasted” (i.e. cannot be used for “latency hiding”) because they cannot be used to advance any warp in computation (during that cycle).

This can be illustrated by the following example: 128 threads (in 4 warps of 32) all execute the same kernel (with instruction string “LLC”) on a single streaming multiprocessor. Figure 2 presents one possible schedule (which is work-conserving). Note that during the first 5 clock cycles, the multiprocessor has a throughput of only one instruction per warp per cycle (Figure 2), because initially *all* warps need to perform two consecutive load/store instructions and the CUDA cores are of no use to any of them (hence remain idle).

Accordingly, our pessimistic makespan derivation assumes that for every instruction of a given warp, *all* other warps are also competing for the same computational unit, at the time of its issue. To enforce this (very pessimistic) assumption, we no longer consider the actual kernel instruction string but rather just the number of instructions of a given type in that string.

Assume that the kernel instruction string α has length I and that there are two types of computational units: load/store and CUDA (represented by “L” and “C” in the string). Then, I_L and I_C is the number of “L”s and “C”s in the kernel instruction string (i.e. $I_L + I_C = I$). From the original kernel instruction string, we derive two strings: one string α_L consisting exclusively of “L”s (I_L in count) and one string α_C consisting exclusively of “C”s (I_C in count). In equations:

$$\alpha_L = \{\underbrace{L\ L\ \ldots\ L}_{I_L\ \text{“L”s}}\} \quad (3)$$

$$\alpha_C = \{\underbrace{C\ C\ \ldots\ C}_{I_C\ \text{“C”s}}\} \quad (4)$$

The pessimistic worst-case makespan is then derived as

$$T = T_L + T_C \quad (5)$$

where T_L is the worst-case makespan for a group of W hypothetical warps executing α_L as kernel (and likewise for T_C and α_C). In turn, T_L and T_C are derived as:

$$T_L = \left\lceil \frac{W}{\sigma_L} \right\rceil I_L \quad (6)$$

$$T_C = \left\lceil \frac{W}{\sigma_C} \right\rceil I_C \quad (7)$$

5. ILP derivation

In this section we present the formulation of the worst-case makespan derivation problem as a binary ILP. The solution of the ILP instance provides the exact (subject to our simplifying assumption) worst-case makespan. In order to generate the ILP instance from the problem instance, we also employ the pessimistic makespan derivation described in the previous section.

Assume that the kernel (known beforehand) consists of I instructions. We can present the sequence of the instructions using binary constants with index $i = 1..I$.

$$IL_i = \begin{cases} 1 & \text{if instruction } i \text{ is for a load/store unit;} \\ 0 & \text{otherwise.} \end{cases} \quad (8)$$

$$IC_i = \begin{cases} 1 & \text{if instruction } i \text{ is for a CUDA core;} \\ 0 & \text{otherwise.} \end{cases} \quad (9)$$

$$\forall i \quad IL_i + IC_i = 1 \quad (10)$$

It is obvious that the schedule for which the worst-case (i.e. longest) makespan is observed can be no longer than T clock cycles, where T is the makespan estimate (5) computed under the simple pessimistic approach described earlier in Section 4.

To describe the schedule of W warps over T clock cycles, we introduce the following binary decision variables, specifying the usage of the resources of the streaming multiprocessor:

$$LS_{w,i,t} = \begin{cases} 1 & \text{if warp } w \text{ performs instruction } i \text{ on} \\ & \text{load/store unit at clock cycle } t\ ; \\ 0 & \text{otherwise.} \end{cases}$$

$$CC_{w,i,t} = \begin{cases} 1 & \text{if warp } w \text{ performs instruction } i \text{ on} \\ & \text{CUDA core at clock cycle } t\ ; \\ 0 & \text{otherwise.} \end{cases}$$

where indexes $w = 1..W$ and $t = 1..T$ stand for warps and clock cycles respectively. Note that $\forall w, i, t$, it holds that $LS_{w,i,t}$ and $CC_{w,i,t}$ cannot both be non-zero (because any instruction can only use a specific type of computational unit).

With the help of these variables, the formulation of the ILP is presented as follows: in subsection 5.1 we derive the objective function, corresponding to the worst-case makespan; subsection 5.2 formulates capacity constraints on the computational resources of the single streaming multiprocessor; subsection 5.3 states precedence constraints for the instructions of the kernel instruction string; subsection 5.4 dwells on constructing constraints that guarantee the work-conserving property of the schedule; subsection 5.5 presents the entire ILP derivation in one place; finally, subsection 5.6 addresses tractability issues, related to solving the optimization problem.

5.1. Objective function

The objective function should be designed in such a way, to provide the longest possible makespan when all the constraints are satisfied. Relying on the precedence constraints between instructions, we notice that the makespan is maximized iff the last instruction of the last warp to complete (whichever that is), is executed as late as possible. Since this would be the I^{th} kernel instruction, the worst-case makespan is then given by

$$\max_{w=1..W,t=1..T}\{t\cdot(LS_{w,I,t}+CC_{w,I,t})\} \tag{11}$$

Given that the objective function of our optimization problem should be linear, we need to add some extra constraints to present (11) in a proper way. Although, in principle, we are not interested in which one of the W warps executes the last instruction in the schedule, specifying that would allow us to simplify (11). Without loss of generality, since all warps are identical, any schedule with worst-case makespan can be transformed into a schedule where the last completing warp is the warp W (e.g. via re-indexing of warps). We can express this additional requirement using $(W-1)$ constraints:

$$\forall w=1..(W-1)$$

$$\sum_{t=1}^{T}(t\cdot(LS_{w,I,t}+CC_{w,I,t}))\le\sum_{t=1}^{T}(t\cdot(LS_{W,I,t}+CC_{W,I,t}))$$

Therefore (11) could be presented as finding the clock cycle when the warp W executes an instruction with the index I.

$$\max_{t=1..T}\{t\cdot(LS_{W,I,t}+CC_{W,I,t})\} \tag{12}$$

However, there exists only one $t'\in[1..T]$ such that warp W performs instruction I at cycle t'. Therefore $\forall t''\in[1..T]$, $t''\neq t'$: $LS_{W,I,t''}=0$ and $CC_{W,I,t''}=0 \Rightarrow LS_{W,I,t''}+CC_{W,I,t''}=0$. Hence expression (12) can be rewritten as a linear function of $LS_{W,I,t}$ and $CC_{W,I,t}$ as follows:

$$\sum_{t=1}^{T}(t\cdot(LS_{W,I,t}+CC_{W,I,t})) \tag{13}$$

This is the objective function (that should be maximized) in our binary ILP-formulation.

5.2. Capacity constraints

As explained in Section 1, the makespan is dependent on how internal resources in a streaming multiprocessor (CUDA cores and load/store units in our case) are shared between threads. Although streaming multiprocessors of modern GPUs have many computational units, these are still finite resources. Additionally, the number of computational units of each type (i.e. L and C) is typically different. Such limitations, among others, can be represented by the following constraints:

An upper bound on the number of load/store instructions that could be performed within a single clock cycle t, could be expressed as:

$$\forall t \quad \sum_{w=1}^{W}\sum_{i=1}^{I}LS_{w,i,t}\le\sigma_L \tag{14}$$

Similarly for the number of CUDA instructions:

$$\forall t \quad \sum_{w=1}^{W}\sum_{i=1}^{I}CC_{w,i,t}\le\sigma_C \tag{15}$$

Any warp is able to perform no more than one instruction at a single clock cycle:

$$\forall w,t \quad \sum_{i=1}^{I}LS_{w,i,t}\le 1,\quad \sum_{i=1}^{I}CC_{w,i,t}\le 1 \tag{16}$$

Any instruction can only be executed on a computational unit of a specific respective type:

$$\forall w,i \quad \sum_{t=1}^{T}LS_{w,i,t}=IL_i,\quad \sum_{t=1}^{T}CC_{w,i,t}=IC_i \tag{17}$$

The constraints expressed by Equations (10) and (17) mean that:
– If $(IC_i=1)$ then $(\forall w,t \quad LS_{w,i,t}=0)$
– If $(IL_i=1)$ then $(\forall w,t \quad CC_{w,i,t}=0)$
Additionally, Equations (17) and (10) ensure that every instruction is performed by every warp.

5.3. Precedence constraints

Since the kernel instructions are executed in a particular order by all warps, we must model the constraints of precedence between them. For these purposes it is useful to introduce auxiliary (not decision) variable $Y_{w,i}$ which denotes the clock cycle when warp w executes instruction i. This new variable facilitates expressing the constraint that $\forall i=1..(I-1)$ and for every warp, the instruction $i+1$ cannot be executed until after the instruction i has been executed by the same warp:

$$\forall w \quad Y_{w,1}<Y_{w,2}<\cdots<Y_{w,I-1}<Y_{w,I} \tag{18}$$

Taking into account Equations (17) and (10), one may see that $(Y_{w,i}=t)$ is equivalent to $(\sum_{t'=1}^{t}(LS_{w,i,t'}+CC_{w,i,t'})=1)$

That could be written as

$$Y_{w,i}=\sum_{t=1}^{T}(t\cdot(LS_{w,i,t}+CC_{w,i,t})) \tag{19}$$

By substitution of Equation (19) to (18), we get:

$$\forall w,i=1..(I-1)$$

$$\sum_{t=1}^{T}(t\cdot(LS_{w,i,t}+CC_{w,i,t}))<\sum_{t=1}^{T}(t\cdot(LS_{w,i+1,t}+CC_{w,i+1,t}))$$

In linear programs the inequalities should be non-strict [21]. Therefore (since the decision variables are integer) we rewrite the above as:

$$\forall w,i=1..(I-1)$$

$$1+\sum_{t=1}^{T}(t\cdot(LS_{w,i,t}+CC_{w,i,t})) \leq \sum_{t=1}^{T}(t\cdot(LS_{w,i+1,t}+CC_{w,i+1,t}))$$

5.4. Work-conserving constraints

One of our assumptions, stated in Section 2, was about the scheduling policy implemented in GPU. Namely, that it is work-conserving. This means that whenever there are warps available and free computational resources on the streaming multiprocessor, the scheduler must select some warp for execution. Next, we introduce some additional variables, for the purpose of modelling the work-conserving property of the schedule via ILP constraints.

Let us assume that instruction i is for a load/store unit ($IL_i = 1, IC_i = 0$). Then $LSREADY_{w,i,t} = 1$ iff warp w was ready to execute instruction i at clock cycle t (i.e. it had already executed instructions $1..(i-1)$) but did not. Similarly with variable $CCREADY_{w,i,t}$ if $IL_i = 0$ and $IC_i = 1$. In formal notation:

$$\forall w, t$$

$$LSREADY_{w,1,t} = \begin{cases} 1 & \text{if } (IL_1 = 1) \wedge (t < Y_{w,1}) \ ; \\ 0 & \text{otherwise.} \end{cases}$$

$$CCREADY_{w,1,t} = \begin{cases} 1 & \text{if } (IC_1 = 1) \wedge (t < Y_{w,1}) \ ; \\ 0 & \text{otherwise.} \end{cases}$$

$$\forall w, i = 2..I, t$$

$$LSREADY_{w,i,t} = \begin{cases} 1 & \text{if } (Y_{w,i-1} < t) \wedge (IL_i = 1) \\ & \wedge (t < Y_{w,i}) \ ; \\ 0 & \text{otherwise.} \end{cases}$$

$$CCREADY_{w,i,t} = \begin{cases} 1 & \text{if } (Y_{w,i-1} < t) \wedge (IC_i = 1) \\ & \wedge (t < Y_{w,i}) \ ; \\ 0 & \text{otherwise.} \end{cases}$$

A schedule is not work-conserving iff there exists some warp w that is ready to perform some instruction i at clock cycle t, but stays idle, even if there were spare computational units (of the type that instruction i runs on). This scenario could be expressed as follows:

$$\begin{aligned} \exists w, t \quad & ((\sum_{i=1}^{I} LSREADY_{w,i,t} \neq 0) \wedge \\ & (\sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} < \sigma_L)) \vee \\ & ((\sum_{i=1}^{I} CCREADY_{w,i,t} \neq 0) \wedge \\ & (\sum_{w'=1}^{W}\sum_{i=1}^{I} CC_{w',i,t} < \sigma_C)) \end{aligned} \tag{20}$$

If and only if the expression (20) does not hold (or equivalently, its logical complement holds), the schedule is work-conserving. The logical complement to (20) can be derived via application of De Morgan's laws and is the following:

$$\begin{aligned} \forall w, t \quad & ((\sum_{i=1}^{I} LSREADY_{w,i,t} = 0) \vee \\ & (\sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} = \sigma_L)) \wedge \\ & ((\sum_{i=1}^{I} CCREADY_{w,i,t} = 0) \vee \\ & (\sum_{w'=1}^{W}\sum_{i=1}^{I} CC_{w',i,t} = \sigma_C)) \end{aligned} \tag{21}$$

In a system of ILP-constraints, expression (21) can be split into two constraints that make the following boolean expressions true:

$$\forall w, t$$

$$((\sum_{i=1}^{I} LSREADY_{w,i,t} = 0) \vee (\sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} = \sigma_L)) \tag{22}$$

and

$$\forall w, t$$

$$((\sum_{i=1}^{I} CCREADY_{w,i,t} = 0) \vee (\sum_{w'=1}^{W}\sum_{i=1}^{I} CC_{w',i,t} = \sigma_C)) \tag{23}$$

Let us consider constraint (22). The equality

$$\sum_{i=1}^{I} LSREADY_{w,i,t} = 0 \tag{24}$$

holds iff $\forall i \quad LSREADY_{w,i,t} = 0$.

From the definition, we know that $LSREADY_{w,i,t} = 0$ iff the following boolean expressions hold:

$$\neg((IL_1 = 1) \wedge (t < Y_{w,1})) = true \tag{25}$$

for $LSREADY_{w,1,t} = 0$;

$$\neg((Y_{w,i-1} < t) \wedge (IL_i = 1) \wedge (t < Y_{w,i})) = true \tag{26}$$

for $LSREADY_{w,i,t} = 0 \quad \forall i = 2..I$.

Expressions (25) and (26) can be equivalently rewritten as:

$$(IL_1 = 0) \vee (t \geq Y_{w,1}) = true \tag{27}$$

for $LSREADY_{w,1,t} = 0$;

$$(Y_{w,i-1} \geq t) \vee (IL_i = 0) \vee (t \geq Y_{w,i}) = true \tag{28}$$

for $LSREADY_{w,i,t} = 0 \quad \forall i = 2..I$.

Taking into account that

$$\sum_{t'=1}^{t}(LS_{w,i,t'} + CC_{w,i,t'}) = \begin{cases} 1 & \text{if } t \geq Y_{w,i}; \\ 0 & \text{otherwise.} \end{cases}$$

and

$$\sum_{t'=t}^{T}(LS_{w,i,t'}+CC_{w,i,t'})=\begin{cases}1 & \text{if } Y_{w,i}\geq t;\\ 0 & \text{otherwise.}\end{cases}$$

we can rewrite the left hand sides of boolean expressions (27) and (28) as

$$TL_{w,1,t}=(IL_1=0)\vee(\sum_{t'=1}^{t}(LS_{w,1,t'}+CC_{w,1,t'})=1)$$

and

$$\begin{aligned}TL_{w,i,t}=&(\sum_{t'=t}^{T}(LS_{w,i-1,t'}+CC_{w,i-1,t'})=1)\vee(IL_i=0)\vee\\&(\sum_{t'=1}^{t}(LS_{w,i,t'}+CC_{w,i,t'})=1)\quad\forall i=2..I\end{aligned}$$

respectively (using the shorthand $TL_{w,i,t}$ for the purpose of making equations more readable).

In such a way the equality (24) can be equivalently rewritten as:

$$TL_{w,1,t}\wedge TL_{w,2,t}\wedge\cdots\wedge TL_{w,I,t}=true \tag{29}$$

To express $\sum_{w'=1}^{W}\sum_{i=1}^{I}LS_{w',i,t}=\sigma_L$, which is the right hand side part of (22), let us denote

$$E_t=\begin{cases}1 & \text{if } \sum_{w'=1}^{W}\sum_{i=1}^{I}LS_{w',i,t}=\sigma_L\ ;\\ 0 & \text{otherwise.}\end{cases}$$

Intuitively, $E_t=1$ iff there is no spare capacity of load/store units in the streaming multiprocessor at clock cycle t. An equivalent (but more convenient) definition of the above binary decision variable is:

$$E_t=1-sign(\sigma_L-\sum_{w'=1}^{W}\sum_{i=1}^{I}LS_{w',i,t}) \tag{30}$$

where

$$sign(r)=\begin{cases}1 & \text{for } r>0;\\ 0 & \text{for } r=0;\\ -1 & \text{for } r<0.\end{cases}$$

Subject to (29) and the definition of E_t, (22) is rewritten as:

$$(TL_{w,1,t}\wedge TL_{w,2,t}\wedge\cdots\wedge TL_{w,I,t})\vee E_t \tag{31}$$

or equivalently

$$(TL_{w,1,t}\vee E_t)\wedge(TL_{w,2,t}\vee E_t)\wedge\cdots\wedge(TL_{w,I,t}\vee E_t) \tag{32}$$

We expressed the work-conserving property for load/store units through the boolean expressions presented above. To ensure that these expressions hold, we have to model them using linear constraints. According to Theorems 1 and 2 (see Appendix in [3]), expression (31) can be represented by a single relatively long linear constraint:

$$\begin{aligned}\forall w,t\quad&\frac{1}{2}((-\frac{I-1}{I}+\frac{1}{I}\sum_{i=1}^{I}TL_{w,i,t})+E_t)\leq\\&(TL_{w,1,t}\wedge TL_{w,2,t}\wedge\cdots\wedge TL_{w,I,t})\vee E_t\leq\\&\frac{1}{I}\sum_{i=1}^{I}TL_{w,i,t}+E_t\end{aligned} \tag{33}$$

wherein the boolean expression is treated as an integer (0/1). Similarly expression (32) could be represented by I relatively short linear constraints:

$$\forall w,i,t\quad\frac{1}{2}(TL_{w,i,t}+E_t)\leq TL_{w,i,t}\vee E_t\leq TL_{w,i,t}+E_t \tag{34}$$

Applying a similar approach to (23), using shorthand $TC_{w,1,t}$, where

$$TC_{w,1,t}=(IC_1=0)\vee(\sum_{t'=1}^{t}(LS_{w,1,t'}+CC_{w,1,t'})=1)$$

$$\begin{aligned}TC_{w,i,t}=&(\sum_{t'=t}^{T}(LS_{w,i-1,t'}+CC_{w,i-1,t'})=1)\vee(IC_i=0)\vee\\&(\sum_{t'=1}^{t}(LS_{w,i,t'}+CC_{w,i,t'})=1)\quad\forall i=2..I\end{aligned}$$

and binary decision variable

$$G_t=1-sign(\sigma_C-\sum_{w'=1}^{W}\sum_{i=1}^{I}CC_{w',i,t}) \tag{35}$$

we can present (23) by a single long constraint:

$$\begin{aligned}&\forall w,t\\&\frac{1}{2}((-\frac{I-1}{I}+\frac{1}{I}\sum_{i=1}^{I}TC_{w,i,t})+G_t)\leq\\&(TC_{w,1,t}\wedge TC_{w,2,t}\wedge\cdots\wedge TC_{w,I,t})\vee G_t\leq\\&\frac{1}{I}\sum_{i=1}^{I}TC_{w,i,t}+G_t\end{aligned} \tag{36}$$

or by I short linear constraints:

$$\forall w,i,t\quad\frac{1}{2}(TC_{w,i,t}+G_t)\leq TC_{w,i,t}\vee G_t\leq TC_{w,i,t}+G_t \tag{37}$$

At this point, let us focus on how to model decision variables E_t and G_t (which have non-linear definitions) as linear expressions. By inspecting Equations (30) and (35), we can notice that function $sign()$ takes only non-negative arguments there. In the case of E_t, it is because $\forall t\quad \sigma_L\geq\sum_{w'=1}^{W}\sum_{i=1}^{I}LS_{w',i,t}$ In particular:
if $\sigma_L=\sum_{w'=1}^{W}\sum_{i=1}^{I}LS_{w',i,t}$, then
$sign(\sigma_L-\sum_{w'=1}^{W}\sum_{i=1}^{I}LS_{w',i,t})=sign(0)=0$;
if $\sigma_L>\sum_{w'=1}^{W}\sum_{i=1}^{I}LS_{w',i,t}$, then
$sign(\sigma_L-\sum_{w'=1}^{W}\sum_{i=1}^{I}LS_{w',i,t})=1$.

Since there is no need to "implement" $sign()$ for negative arguments, we can model it as follows. Let us denote the shorthand $SL_t=sign(\sigma_L-\sum_{w'=1}^{W}\sum_{i=1}^{I}LS_{w',i,t})$. The constraint

$$SL_t\leq\sigma_L-\sum_{w'=1}^{W}\sum_{i=1}^{I}LS_{w',i,t} \tag{38}$$

maximize $\sum_{t=1}^{T}(t \cdot (LS_{W,I,t} + CC_{W,I,t}))$ subject to		
iterated variables	expression for constraint	number of constraints
$\forall t$	$\sum_{w=1}^{W}\sum_{i=1}^{I} LS_{w,i,t} \leq \sigma_L$	T
$\forall t$	$\sum_{w=1}^{W}\sum_{i=1}^{I} CC_{w,i,t} \leq \sigma_C$	T
$\forall w = 1..(W-1)$	$\sum_{t=1}^{T}(t \cdot (LS_{w,I,t} + CC_{w,I,t})) \leq \sum_{t=1}^{T}(t \cdot (LS_{W,I,t} + CC_{W,I,t}))$	$W-1$
$\forall w, t$	$\sum_{i=1}^{I} LS_{w,i,t} \leq 1$	$W \cdot T$
$\forall w, t$	$\sum_{i=1}^{I} CC_{w,i,t} \leq 1$	$W \cdot T$
$\forall w, i$	$\sum_{t=1}^{T} LS_{w,i,t} = IL_i$	$W \cdot I$
$\forall w, i$	$\sum_{t=1}^{T} CC_{w,i,t} = IC_i$	$W \cdot I$
$\forall w, i = 1..(I-1)$	$1 + \sum_{t=1}^{T}(t \cdot (LS_{w,i,t} + CC_{w,i,t})) \leq \sum_{t=1}^{T}(t \cdot (LS_{w,i+1,t} + CC_{w,i+1,t}))$	$W \cdot (I-1)$
$\forall t$	$E_t \geq 1 - \sigma_L + \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}$	T
$\forall t$	$E_t \cdot \sigma_L \leq \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}$	T
$\forall w, i, t$	$\frac{1}{2}(TL_{w,i,t} + E_t) \leq TL_{w,i,t} \lor E_t \leq TL_{w,i,t} + E_t$	$W \cdot I \cdot T$
$\forall t$	$G_t \geq 1 - \sigma_C + \sum_{w'=1}^{W}\sum_{i=1}^{I} CC_{w',i,t}$	T
$\forall t$	$G_t \cdot \sigma_C \leq \sum_{w'=1}^{W}\sum_{i=1}^{I} CC_{w',i,t}$	T
$\forall w, i, t$	$\frac{1}{2}(TC_{w,i,t} + G_t) \leq TC_{w,i,t} \lor G_t \leq TC_{w,i,t} + G_t$	$W \cdot I \cdot T$

Fig. 3. The complete ILP formulation (using short constraints)

states the first basic property of the function (that its value cannot be greater than its argument). The second fundamental property (that the value of the function denotes the sign of the argument) is stated by the following constraint:

$$\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} \leq SL_t \cdot (\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t})$$

Without loss of correctness, we can rewrite this as

$$\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} \leq SL_t \cdot \sigma_L \qquad (39)$$

reducing computational complexity for the software implementation.

According to Equation (30) and the definition of SL_t, we can compute E_t as $(1 - SL_t)$, but it will be more efficient to model E_t directly (using the previous derivation of SL_t). Multiplying (38) by (-1) and adding 1 to both sides yields

$$1 - SL_t \geq 1 - (\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t})$$

This can be rewritten as follows:

$$E_t \geq 1 - \sigma_L + \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} \qquad (40)$$

Multiplying (39) by (-1) we get

$$-(\sigma_L - \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t}) \geq (1 - SL_t) \cdot \sigma_L - \sigma_L$$

One may then reduce it to

$$E_t \cdot \sigma_L \leq \sum_{w'=1}^{W}\sum_{i=1}^{I} LS_{w',i,t} \qquad (41)$$

By analogy, for linear constraints (40) and (41) for G_t:

$$G_t \geq 1 - \sigma_C + \sum_{w'=1}^{W} \sum_{i=1}^{I} CC_{w',i,t}$$

$$G_t \cdot \sigma_C \leq \sum_{w'=1}^{W} \sum_{i=1}^{I} CC_{w',i,t}$$

5.5. Summary of the ILP formulation

Let us now present the entire formulation of the binary ILP in one place (Figure 3) (opting for using as short constraints as possible) .

5.6. Resolving the issue of tractability

Integer programming is in common use in various fields [21] and corresponding problems are probably, the most widely-used examples of NP-hard computational problems. Even though bounded ILPs (binary in our case) are, usually, more tractable than those which allow decision variables to take values from infinite domains, it is obvious that even for relatively small number of warps (W), computing the makespan with the ILP formulation presented above will take too much time. However, we can find a marginally pessimistic makespan estimate at only a fraction of the time as follows:

Let $T^{(W)}$ denote the worst-case makespan of W warps (for which we seek an upper bound) and $T^{(x)}$ denote the corresponding worst-case makespan for x warps (with $x \ll W$). By choosing a small enough value for x such that the exact value for $T^{(x)}$ can be tractably computed, according to our ILP derivation, then $T^{(W)}$ can be safely approximated by

$$T^{(W)}_{approx(x)} = \min_{y=1}^{x} T^{(W,y)} \qquad (42)$$

where

$$T^{(W,y)} = \left\lceil \frac{W}{y} \right\rceil \cdot T^{(y)}, \quad y \in \mathbb{N} \qquad (43)$$

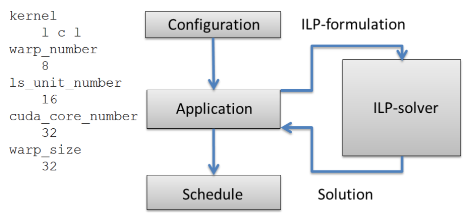
We compute the upper bound on $T^{(W)}$ using Equation (42) and not as $T^{(W,x)}$ because, although $T^{(W,x)}$ typically decreases with increasing x, sometimes there are small increases (especially for very small x). The estimate $T^{(W)}_{approx(x)}$ for $T^{(W)}$ given by (42) improves with higher x, at the cost of rapidly increasing computation times. However, experimental evidence (see the next section) shows diminishing returns, even past small values of x. In other words, the estimate rapidly converges and even for small x, there is very little pessimism.

6. Experiments

As shown in Section 5 the makespan depends on the number of warps, the kernel instruction string and the hardware (namely, the number of computational units of each type and the warp size). We implemented the techniques introduced in this paper in a cross-platform software application that reads the problem instance from a configuration file (Figure 4), constructs the binary ILP-formulation and launches the proprietary ILP-solver (see [10]). After getting the solution, it presents the worst-case makespan and corresponding schedule (like the one in Figure 2) or alternatively computes an estimate using Equation (43) (if the user does not want to wait too long).

In Subsection 5.4 we stated that there are two alternatives for expressing the work-conserving property: either (i) using $W \cdot I \cdot T$ shorter constraints (34), (37) or (ii) using $W \cdot T$ longer constraints (33), (36). We implemented both options and compared their timings. One such comparison is presented in Figure 5. In our experiments, the first option generally gave shorter computation times.

We also explored how the tractable approximation for $T^{(W)}$ (presented in Subsection 5.6) improves/converges with increased values of the parameter x. Figure 6 and Figure 7 present the results of two such experiments. In general, we



```
kernel
    l c l
warp_number
    8
ls_unit_number
    16
cuda_core_number
    32
warp_size
    32
```

Fig. 4. Typical configuration file and application workflow.

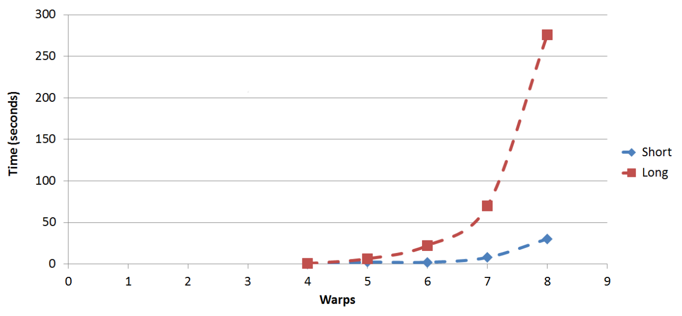


Fig. 5. Computation time for solving ILP-problem with short and long constraints ($\sigma_L = \sigma_C = 1$, "LLCLL")

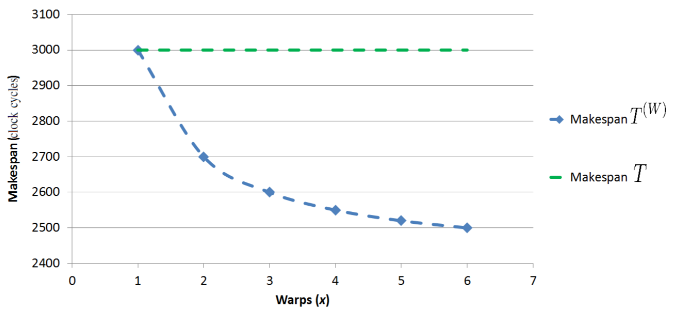


Fig. 6. Convergence of $T^{(W)}$ with increasing x (W=600, σ_L=σ_C=1, "LLCLL"). The horizontal dashed line corresponds to the pessimistic estimate T (Section 4).

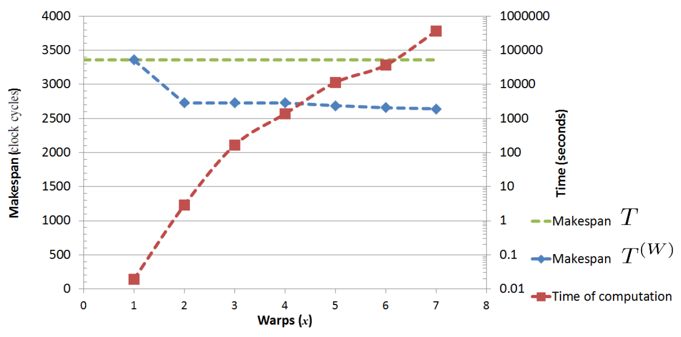


Fig. 7. Growth of computation time and convergence of $T^{(W,x)}$ with increasing x ($W = 420$, $\sigma_L = \frac{1}{2}$, $\sigma_C = 1$, "LCLCL").

observed that the estimate $T^{(W)}$ converges very fast with increasing x and afterwards the improvement to the estimate is minor (diminishing returns). Our interpretation is that this is because the approximation is good even for small values of x. Therefore, although the computation time increases very rapidly with x (Figure 7), one may obtain (i.e. using small x) estimates that are both quite accurate *and* tractably derivable.

7. Conclusion and future work

In this paper we introduce techniques for finding the worst-case makespan for a group of GPU threads: one approach which is pessimistic but has very low computational complexity and another approach (which builds on the former one) which employs Integer Linear Programming for an exact derivation (subject to some simplifying assumptions). Since the exact approach is computationally intractable for a large number of warps, we also introduce a simple way of obtaining, at only a fraction of the time, a safe estimate that is only marginally pessimistic.

In this work, we address a single streaming multiprocessor, based on the NVIDIA Fermi hardware architecture. However, since a GPU contains many streaming multiprocessors, as a next step, we intend to extend our approach to address thread execution over multiple streaming multiprocessors. Doing so will require faithfully modelling how warps are dispatched/partitioned among streaming multiprocessors – something which, to the best of our understanding, is either not fully documented at the moment or subject to change between revisions. Another direction for our future work, will be to relax the assumption about the absence of cache misses. We believe that this line of work will apply also to NVIDIA Kepler and Maxwell [4] – the next generation GPU architectures.

References

[1] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu. An adaptive performance modeling tool for GPU architectures. In *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, January 2010.

[2] J. Balfour. CUDA threads and atomics. Lecture slides (2011) – http://mc.stanford.edu/cgi-bin/images/3/34/Darve_cme343_cuda_3.pdf.

[3] K. Berezovskyi, K. Bletsas, and B. Andersson. Makespan computation for GPU threads running on a single streaming multiprocessor. Technical Report HURRAY-TR-111215, CISTER-ISEP Research Center, Polytechnic Institute of Porto, Available at www.cister.isep.ipp.pt/docs/makespan+computation +for+gpu+threads+running+on+a+single+streaming +multiprocessor/685/view.pdf, December 2011.

[4] B. Dally. GPU computing to exascale and beyond. Lecture slides – http://www.nvidia.com/content/PDF/sc_2010/theater/ Dally_SC10.pdf.

[5] G. Elliott and J. Anderson. Real-time multiprocessor systems with GPUs. In *18th International Conference on Real-Time and Network Systems*, November 2010.

[6] V. Glavtchev, P. Muyan-Özçelik, J. M. Ota, and J. D. Owens. Feature-based speed limit sign detection using a graphics processing unit. In *Intelligent Vehicles Symposium (IV)*, pages 195–200, 2011.

[7] D. Göddeke. GPGPU:basic math tutorial. http://www.mathematik.uni-dortmund.de/

[8] M. Gouiffès, A. Patri, and M. Vasiliu. Robust obstacles detection and tracking using disparity for car driving assistance. In *SPIE*, volume 7539, 2010.

[9] S. Hong and H. Kim. A memory-level and thread-level parallelism aware GPU architecture performance analytical model. In *36th International Symposium on Computer Architecture (ISCA-36)*, June 2009.

[10] IBM Corporation. IBM ILOG CPLEX Optimization Studio. Product brief – http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/modeling/, 2011.

[11] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar. Resource sharing in GPU-accelerated windowing systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011.

[12] S. Kato, K. Lakshmanan, A. Kumar, Y. Ishikawa, and R. Rajkumar. Rgem: A responsive GPGPU execution model for runtime engines. In *32nd IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[13] Khronos OpenCL Working Group. The OpenCL specification. Available online – www.khronos.org/registry/cl/specs/opencl-1.2.pdf#page=49, 2011.

[14] W. Liu, B. Schmidt, and W. Müller-Wittig. Performance analysis of general-purpose computation on commodity graphics hardware: A case study using bioinformatics. *Journal of Signal Processing Systems (JSPS)*, 48(3):209–221, 2007.

[15] NVIDIA Corp. NVIDIA's next generation CUDA compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[16] NVIDIA Corporation. GeForce 256 the world's first GPU. Product brief – http://www.nvidia.com/page/geforce256.html, 1999.

[17] NVIDIA Corporation. NVIDIA CUDA C programming guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/ C/doc/CUDA_C_Programming_Guide.pdf, 2011.

[18] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, pages 80–113, 2007.

[19] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2008.

[20] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem – overview of methods and survey of tools. *ACM Trans. Embedded Computing Systems*, 7(3):1–53, 2008.

[21] L. A. Wolsey and G. L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1998.