CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Faster Makespan Estimation for GPU Threads on a Single Streaming Multiprocessor

Kostiantyn Berezovskyi
Konstantinos Bletsas
Stefan M. Petters

CISTER-TR-130904
Version:
Date: 09-11-2013

Faster Makespan Estimation for GPU Threads on a Single Streaming Multiprocessor

Kostiantyn Berezovskyi, Konstantinos Bletsas, Stefan M. Petters

CISTER Research Unit

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.cister.isep.ipp.pt

Abstract

Graphics Processing Units (GPUs) are widely used to unload the CPUs, liberate other resources of a given computer system, and provide an alternative to multiprocessor computers as a means of processing computationally expensive parallel tasks. The recent trend of utilizing GPUs in embedded systems necessitates the development of timing analysis techniques for finding the joint worst-case execution time for a group of GPU threads of the same parallel application, on a streaming multiprocessor. The state-of-the-art approaches for computing the exact maximum makespan of GPU threads running on a single streaming multiprocessor are intractable and even pessimistic approximations usually take a long time to complete. We therefore develop a technique for finding an estimate of the maximum makespan using metaheuristics. Its simplicity, flexibility and ability for massive parallelization, determine a potential of usage for soft real-time systems.

Faster Makespan Estimation for GPU Threads on a Single Streaming Multiprocessor

Kostiantyn Berezovskyi and Konstantinos Bletsas and Stefan M. Petters
CISTER/INESC-TEC, ISEP
Polytechnic Institute of Porto, Portugal
Email: {kosbe, ksbs, smp}@isep.ipp.pt

***Abstract*—Graphics Processing Units (GPUs) are widely used to reduce the load on CPUs and liberate other resources of a given computer system. The recent trend of utilizing GPUs in embedded systems necessitates the development of timing analysis techniques for finding the joint worst-case execution time for a group of GPU threads of the same parallel application, on a streaming multiprocessor. The state-of-the-art approaches for computing the exact maximum makespan of GPU threads running on a single streaming multiprocessor are computationally expensive and even pessimistic approximations usually take a long time to complete. We therefore develop a technique for finding an estimate of the maximum makespan using metaheuristics. Its simplicity, flexibility and ability for massive parallelization, determine a potential of usage for soft real-time systems.**

I. Introduction

The growing use of Graphics Processing Units (GPUs) as general-purpose processors in embedded systems provides the scientific community with the serious challenge of developing methods for finding the finishing time of GPU threads in the worst-case scenario. Traditional worst-case execution time analysis [25] is not applicable as it assumes single-threaded applications with access to all computational resources. In contrast, GPU applications are structured as multiple concurrent threads competing for the same computational resources. Consequently, we are not interested in one particular thread but in a group of many threads whose joint execution provides the result. Hence, the focus on the worst-case makespan – the longest possible time interval from the moment when the "earliest" thread starts executing, until the "latest" thread terminates.

Previous work [4] dwells on the problem of finding the maximum makespan as an optimization problem, specifically an Integer Linear Problem (ILP). Given that ILP problems in the general case are well-known examples of NP-hard computational problems, finding an exact worst-case makespan may take too long. More tractable methods for makespan estimation are hence an interesting research direction. An upper bound on the makespan can still be found using a pessimistic method also presented in previous work [4], however, it still requires a long analysis time to achieve good accuracy.

Yet, although not underestimating the worst-case makespan is crucial for safety critical systems, for many applications in the area of soft real-time systems (which tolerate a rare missed deadline), a tight lower bound on the worst-case makespan would be acceptable, as an estimate. One of the applications could be eye tracking – the process of measuring the motion of an eye and the gaze-point determination (what is in the focus). Recent developments both in the academia and the industry show great potential of utilizing eye tracking methodologies in medicine [8], driver assistance [24], entertainment systems [23], and as an instrument for scientific researches [16]. Exploiting the GPUs to accelerate such techniques can help realize that potential [7], [17], and estimating the makespan of GPU threads has paramount importance for such usage.

Previous work [4] presented the approaches for computing an exact value (or upper bounds) for the worst-case makespan. However, due to practical limitations (potentially long running times, for long GPU-code), in this work we propose an alternative, more practical technique for estimating the worst-case makespan. Its output is a tight lower bound on the worst-case makespan, which could be useful for soft (not hard) real-time systems, depending on the strictness of the timeliness guarantees required by the corresponding application. For example, although for some soft real-time applications, accounting for the average case achieves a good quality of service (e.g. for a gaming interface, as in [23]), for other soft real-time applications (such as driving decision support [24]) reasonably accurate estimates of the worst case (even if borderline optimistic) are appropriate. Our technique targets the latter kind of applications.

This work focuses on estimating the maximum makespan using *metaheuristics* – computational methods that try to find a better solution for an optimization problem iteratively, and statistically tend to converge to the global optimum over time. The detailed explanation of our approach as well as extensive literature review and implementation details are presented in an associated technical report [5] that is referenced through the paper.

In the remainder of the paper the next section presents some considerations to motivate the idea behind the new technique. Section III discusses the system model. Sections IV and V introduce the proposed metaheuristic. Section VI discusses the generation of suitable initial solutions and aspects of efficient implementation. Section VII provides a case study and some evaluation. Section VIII concludes.

II. Related work

There is growing interest in massively parallel processors in the real-time systems community. We believe (as does Lisper [14]) that high-performance data-parallel tasks in future

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'); by FCT-MCTES and by ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/82069/2011.

real-time embedded systems will be delegated to specialized co-processors, to be run in parallel on many of their cores. In such heterogeneous systems, certain work is still done on the main processor(s) while other work is delegated to the specialized co-processor that is dedicated to that particular type of computations. However, there is no need to idle the CPU waiting for the computation results from a co-processor; while the co-processor is busy with the specific work, the CPU may be used to schedule other entities of computation in the system. This kind of computational arrangement is covered by the *limited parallel model* [13]. In order to apply standard uniprocessor response time analysis techniques to limited parallel systems, one would need to disregard parallelism, leading to unnecessary pessimism. Audsley et al. [3] extended traditional worst-case response time analysis [2], [10], such that execution in hardware is no longer pessimistically treated as interfering with software execution. This involves the identification of a different worst-case scenario than that under traditional analysis. Tighter response times may be derived by considering the actual temporal pattern of execution in software/hardware for each task [6]. These techniques [3] [6], assume that the execution times for hardware-mapped portions of the application are given as input; our approach may be used to provide such input.

Kato et al. [11] present runtime GPU management in the operating system space with emphasis on a non-preemptive GPU-kernel execution and a virtualization. Elliott et al. [9] consider GPUs as shared resources. Their GPU management framework contains, among other elements, an execution cost predictor responsible for estimating the execution time of the real-time jobs. However, the estimation is based on the past behaviour of the jobs, hence, our technique can complement such systems by providing the makespan based on an analytical approach instead. Mangharam et al. [15] discussed the runtime scheduling of anytime algorithms for real-time systems. The estimation of the GPU-kernel execution time is still derived from empirical results but their schedulers are designed to adapt to the variations in actual execution time. Our approach can provide a precise worst-case estimate of such systems and thereby decrease the overhead induced by the scheduler while it changes the execution parameters to fit the actual amount of time left.

III. System model

Our analysis considers a streaming multiprocessor inspired by NVIDIA Kepler [20] and NVIDIA Fermi [18] – hardware architectures of GPUs, capable not only of rendering graphics but also of performing general-purpose computations with the help of a specialized programming environment called Compute Unified Device Architecture (CUDA) [21]. These architectures include multiple so-called *streaming multiprocessors* (Figure 1(a)) and a shared on-chip memory.

A. Streaming multiprocessor

Each streaming multiprocessor has a relatively complex structure, which makes its timing analysis a non-trivial open problem. Therefore, in this paper, we restrict our focus to the timing analysis of a single such streaming multiprocessor. This is a necessary first step, before we can address (as future work) the timing analysis of the entire GPU which

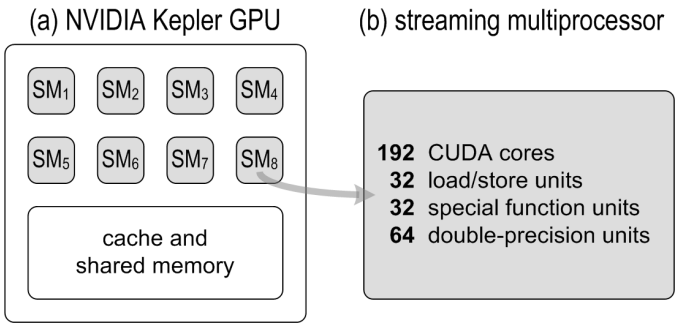

Fig. 1. An NVIDIA Kepler GPU, comprising 8 streaming multiprocessors. Each streaming multiprocessor has a multitude of CUDA cores and load/store, special function and double precision units.

contains multiple streaming multiprocessors, contending for the memory subsystem.

The streaming multiprocessor (Figure 1(b)) includes (i) multiple CUDA cores capable of boolean, integer and floating-point arithmetic, (ii) multiple "load/store" units that load data from/store data to cache or DRAM, (iii) multiple special function units implementing sine, cosine, square root and boolean inversion computation directly in hardware and (iv) multiple double-precision units for 64-bit arithmetic. GPUs evolve fast (even by chip makers' industry standards) hence the configuration of the streaming multiprocessors of different GPU models often varies (although these GPUs could belong to the same generic GPU architecture). The configuration of a GPU-chip is mainly determined by the number of streaming multiprocessors and the number of computational units of each type in each of those. It varies in GPUs of different version ("compute capability", in NVIDIA parlance [21]). For example, for the GPU device of compute capability 2.0 the streaming multiprocessor includes 32 CUDA cores and 16 load/store units. Therefore it is possible that 32 threads perform arithmetic operations concurrently but only 16 threads can issue load/stores in parallel.

For the sake of simplifying the analysis, in our model of a streaming multiprocessor we assume no pipelining in computational units. That is the computational unit of any kind is a mono-element that is fully utilized to perform a respective instruction. As opposed to the computational units under our model, pipelined unit would comprise a sequence of processing elements such, that an output of one element is the input for the next element in the sequence. Our non-pipelining assumption comes at the cost of the pessimism of dropping the instruction throughput by not having instruction level parallelism in case when there are enough consecutive instructions to be performed by the same computational unit.

B. Entities of computation

GPUs were traditionally designed for graphics, consequently they are optimized for running a large number of threads. It is the threads' joint execution (not that of any individual thread) which gives the result. This is also why the hardware architectures under consideration define a more

coarse-grained entity of scheduling, the *warp* – a group of GPU threads that execute the same kernel and in most cases perform the same instruction concurrently. A warp is processed most efficiently when all of its threads, together in parallel, follow the same execution path. However, every individual thread has its own execution context (instruction address counter, states of registers, etc.), therefore it is able to execute and branch independently of other threads within the same warp. Since warps execute independently of each other, regardless of whether they are taking the same path or not, talking about control flow divergence only makes sense for threads within a single warp. If the threads of the same warp branch in different directions, the hardware sequencer keeps track of the diverged threads. It broadcasts the instruction fetch to the computational units that serve the threads of the same branch. Upon reaching the point of convergence, the threads stall waiting for the threads of the other branch, so that they can resume the execution of a common instruction together in parallel once again.

A streaming multiprocessor manages, schedules, and executes warps. The scheduling engine of a streaming multiprocessor comprises several *warp-schedulers* each of which includes a few instruction *dispatch units*. Given warps to execute, a streaming multiprocessor allocates them among its warp-schedulers. Then, at instruction issue time, each warp-scheduler selects an active warp (one that has threads ready to execute its next instruction) and issues a few independent instructions from corresponding threads. The number of the instructions that a warp scheduler can issue for the corresponding active warp within a given cycle is bounded by the number of instruction dispatch units in the warp-scheduler but is also subject of the availability of free computational units to process those instructions. Therefore, if the warp-scheduler includes δ instruction dispatch units, up to δ instructions (with no dependencies between each other) could be performed concurrently, if computational units are available.

C. Simplifying assumptions

The execution context of any thread is stored in the on-chip memory [21] as long as the corresponding warp exists, therefore switching from one context to another is lightweight. The term *instruction latency* specifies the number of clock cycles it takes for a warp to execute a given instruction. Full utilization of the streaming multiprocessor is achieved when there is enough workload to keep all its computational units continuously busy. For example, when a warp is stalled on I/O, the streaming multiprocessor quickly switches to another warp (in a single cycle). This technique is known as "latency hiding". However, to simplify the analysis, we assume that all I/O is served by the local caches ("always hit") and that there is no off-chip data traffic (to main memory). Hence, any load/store instruction always takes a single clock cycle. This optimistic assumption (to be relaxed in future work) is partially justified by the fact that, there is a decent amount (by co-processors' standards) of the on-chip memory in the GPU architectures under consideration [18], [20]. Another simplifying assumption is that all (other) instructions take a fixed respective number of clock cycles (which depends on the type of instruction).

As stated earlier, warps compete for the computational resources of a streaming multiprocessor according to some undocumented scheduling policy. The chip-maker has reported [19] about the move away from complex scheduling logic implemented in hardware (as done in NVIDIA Fermi) towards software scheduling that is performed at run time (in NVIDIA Kepler). However, we still do not have concrete publicly available information about the actual scheduling policy. As in previous work [4], we therefore simply assume that the scheduling is work-conserving: whenever there are warps available and free computational units in a streaming multiprocessor, these units are used to execute some warps. This is a conservative approach, because although the actual internal scheduling policy (whichever that is) probably promotes processing efficiency by doing lookahead scheduling, our search for the worst case will assume it to be more inefficient than it is.

There exist $\sigma{=}4$ warp-schedulers inside each streaming multiprocessor in NVIDIA Kepler [20] and, in turn, each warp-scheduler contains a pair of instruction dispatch units. This pair may issue up to two instructions of the same warp (with no dependencies between each other) to execute in parallel. However, because of the lack of detailed documentation about the semantics of intra-warp parallelism, in this work, we pessimistically assume that each warp scheduler only ever uses one of its two instruction dispatch units. Relaxing this assumption, subject to extracting some information about the semantics of intra-warp parallelism, is also left for future work.

D. Kernel instruction string

CUDA not only provides users with the APIs for high-level programming languages (C, C++, Fortran, wrappers for Java and Python), support for computational interfaces (OpenCL, DirectCompute) and for directive-based OpenACC, but it also provides a virtual Instruction Set Architecture (ISA) which is kept relatively stable over the generations of the GPUs developed by NVIDIA. This ISA, the corresponding pseudo-assembly language and the low-level virtual machine are all called PTX as they were designed for *parallel thread execution*. The high-level GPU-code is processed by a specialized compiler (which supports the extensions that CUDA adds to programming languages); the one from NVIDIA is called `nvcc` [21]. Running this compiler with the `-ptx` flag will output the human-readable representation of the pseudo-assembly code that is put into an object file. This file serves as input to the CUDA-driver which includes another compiler that translates the PTX-code into the target ISA – a binary code that can be run on a particular hardware. Although PTX-code is not the machine code that is actually executed by the hardware, we (like Ryoo et al. [22]) rely on it for the purposes of counting the number of the instructions and their mix. Given that we are interested in the usage of the computational units of a streaming multiprocessor, we abstract away from the assembly code using the kernel instruction string [4] – a sequence of "L", "C", "S", and "D" symbols, each of which represents a hardware instruction that should be performed on load/store unit ("L"-instruction), CUDA-core

The term "warp" is NVIDIA terminology. AMD ATI GPUs have a similar concept to that of a warp called a "wavefront" [1].

To the best of our understanding, since it is not clear in the documentation, these are consecutive instructions.

("C"-instruction), special function unit ("S"-instruction) and double-precision 64-bit unit ("D"-instruction). For example, the kernel instruction string "LC" specifies that an instruction should be carried out by the load/store unit, followed by an instruction for a CUDA core.

Although modelling assembly code with such a short alphabet is a simplifying abstraction when compared to the PTX representation, the focus of this model is to represent how the GPU threads share the computational units of a streaming multiprocessor subject to constraints (capacity, precedence, etc.). We assume that instructions are dispatched according to the order presented in the kernel instruction string and there are no dependencies between consecutive instructions.

E. Modelling the streaming multiprocessor configuration and multi-cycle instructions

In order to address GPUs of different compute capabilities and to make possible adjusting the model to future architectures, we will introduce some parameters specifying the configuration of a streaming multiprocessor. For each distinct type U of computational unit inside a streaming multiprocessor (where U can be C, L, S or D), let

$$\sigma_U = \frac{uUnitsNumber}{warpSize} \tag{1}$$

where $uUnitsNumber$ is the number of U-units in a streaming multiprocessor and $warpSize$ is the number of threads per warp. Then, σ_U specifies the maximum number of warps that may be executing a "U"-instruction within the same clock cycle on a single streaming multiprocessor.

Note that it may be the case that $\sigma_U < 1$. For example, in compute capability 2.0 devices the warp size is 32 but the load/store units are half as many. In such cases, 16 threads of the warp (a *half-warp* [21]) execute an "L"-instruction in one clock cycle, and the other half-warp executes this instruction in a later cycle. Schedule-wise, this is akin to having $\sigma_L = 1$ and a two-cycle latency for the load/store instruction. Such behaviour, however, cannot be directly modelled by our technique (described from Section IV onwards), which works with single-cycle latencies. Therefore, to accommodate the case of fractional σ_u and also multi-cycle instruction latencies (typical of D- and S-instructions) in a unified manner we introduce the following modelling transformation:

Let us assume that the number of computational units of a given type is a power of 2 (as is the warp size), as is typical in NVIDIA general-purpose GPU-architectures – the only exception being devices of compute capability 2.1. Then:

- If $\sigma_U < 1$, we assume that $\sigma_U = 1$ and replace each "U" in the kernel instruction string with $\frac{1}{\sigma_U}$ "U"s.
- If an instruction of type U takes $x \geq 2$ cycles, we replace each "U" in the kernel instruction string with x "U"s.

These transformations can be applied in a combined manner to the same instruction. For example, if a device has 16 type-U units and the respective latency of a U-instruction is 4 cycles (with a warp size of 32), then after application of the transformation, each "U" in the kernel instruction string is replaced by $(32/16) \cdot 4 = 8$ "U"s (of a notional single-cycle latency) and $\sigma_U = 1$ is assumed.

This transformation, given our earlier pessimistic assumption of no intra-warp parallelism, is safe since it enlarges the solution space from which the worst-case schedule is to be identified. To highlight this, consider an instruction with 2-cycle latency: in practice it uses a computational unit for two consecutive cycles, whereas, with the transformation, these cycles can be spaced apart.

F. Summary

The assumptions and the most important considerations of the section are summarized as follows:

- A streaming multiprocessor includes four types of computational units: load/store, special function, double-precision, CUDA cores.
- We pessimistically assume that computational units of each kind are not pipelined.
- For the purpose of scheduling in parallel the threads are organized into groups of $warpSize$ threads called warps.
- All threads of all warps of a given streaming multiprocessor execute the same kernel instruction string.
- All the data needed are in cache, therefore, we do not have to account the latency of memory operations.
- Any instruction takes a single clock cycle, as typical of most 32-bit CUDA instructions in NVIDIA Kepler [21], executing in "atomic"-fashion – it holds the computational resource exclusively and cannot be interrupted. If necessary, the kernel instruction string is normalized according to the transformation described in Section III-E in order to obtain these semantics, at the cost of some pessimism.
- We assume that there is no out of order instruction dispatch and consecutive instructions are independent.
- The warps are scheduled in a work-conserving way by σ warp-schedulers, and we pessimistically assume that only a single instruction can be scheduled from the given warp by the available warp-scheduler. Therefore, the number of warps that could be processed in parallel by a single streaming multiprocessor is bounded by:

  $$\min\{\sigma, \sigma_L + \sigma_C + \sigma_S + \sigma_D\}$$

  A warp could be scheduled by at most one warp-scheduler at a time.

IV. Warp pseudo-precedence string

To apply metaheuristics to the problem of finding lower bound on the maximum makespan we need to address the following questions: How to represent a solution for metaheuristic? (i) What is the method to evaluate particular solution? (ii) How our problem is formulated in these terms? (iii)

A. Solution representation

For the exact ILP-based approach [4], the objective is to maximize the makespan and the solution of an optimization problem is presented in the form of decision variables. For an approach using metaheuristics the objective remains to find the maximum makespan as well, but a question to consider is how to most conveniently represent the solution. One option is to express the solution in the form of the corresponding schedule as depicted in Figure 2.

A schedule representation not only contains all necessary information, such as the kernel instruction string, warp number, configuration of the streaming multiprocessor, the makespan, but it is also intuitive and readily understandable by humans. Still, we should check how suitable this representation is in the context of a metaheuristic that searches through a large solution space moving iteratively from the current solution to the neighbour solution, both being relatively "close" to each other. Let us apply the concept of the *neighbour solution*, which is the core of the metaheuristics, to a schedule. If we move some instruction of some warp to a different clock cycle in the schedule in Figure 2, we can consider the resulting schedule in Figure 3 as a neighbour solution to the original one.

However, we can notice that in our example in Figure 3, just by moving that single instruction we are breaking the work-conserving property of the scheduling policy (at clock cycle 5 there is spare capacity of load/store units and a pending "L"-instruction for warps with the identifiers 1, 2 and 3, but the streaming multiprocessor is staying idle). This in turn makes the new solution invalid. The verification (regarding the precedence constraints or the work-conserving properties) of the altered schedule would be computationally expensive and there is no straightforward way of generating *a priori* valid schedules by moving instructions, other than validating *a posteriori*.

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

Fig. 2. Possible schedule ($\sigma_L = \sigma_C = 1$) as a valid solution

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

Fig. 3. An invalid solution (the work-conserving property is violated)

Therefore the schedule itself is probably not the best way of representing a solution, when using metaheuristics. For these purposes we therefore invented another data structure: the *warp pseudo-precedence string*. One possible way to derive the warp pseudo-precedence string from a schedule is the following: traversing the cells of the schedule, column by column, from top to bottom, we append to an (initially null) integer string the identifier of the warp that performs some instruction in the corresponding clock cycle. For our example in Figure 2 the warp pseudo-precedence string is the following:

$$1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3 \quad 4 \quad 1 \quad 4 \quad 2 \quad 3 \quad 4 \qquad (2)$$

Let us consider the warp pseudo-precedence string as a solution for the metaheuristics.

B. Solution evaluation

Since we are searching for the longest makespan, we need to compute it for every solution by constructing corresponding schedule. To build a schedule from a warp pseudo-precedence string we simply traverse warp identifiers in the string one by one from left to right, and insert the corresponding instruction by the respective warp in the earliest clock cycle (i.e. in the left-most position in the schedule) possible, subject to capacity and precedence constraints. To determine whether this instruction is for a load/store unit or for a CUDA core etc., we need to keep track of how many instructions by each warp we have already scheduled at any instant. In other words, if we have already scheduled k instructions by the warp in consideration, then we need to examine the $(k+1)^{th}$ instruction of kernel instruction string, to see which computation unit should process it (e.g. if that is "L", or "C" etc.) The simple algorithm is presented in Figure 4.

```
//Warp pseudo-precedence string.
INPUT: warpPrecStr;
OUTPUT: schedule;

while (warpPrecStr is not fully traversed)
    //From warpPrecStr:
    w = read_current_warp_id();

    //According to the kernel instruction string:
    i = read_current_instruction_type_by_warp(w);

    //Subject to capacity and precedence constraints:
    t = find_earliest_cycle_wherein_possible_execute(w, i);

    add_to_schedule(w, i, t);
```

Fig. 4. The algorithm for constructing the schedule.

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

Fig. 5. A valid neighbour solution (with increased makespan)

C. Problem formulation

Hence, we can address the problem of estimating the maximum makespan from the following standpoint: find a warp pseudo-precedence string such that the corresponding makespan is maximized, subject to the configurations of the streaming multiprocessor under consideration.

We can try to get a neighbour solution by swapping the positions of warp identifiers in the string (2). There are many possible ways to do that, but let us consider moving all the identifiers of the warp 4 to the end of the string. After doing that, the warp pseudo-precedence string becomes "1 1 2 2 3 3 1 2 3 4 4 4". The schedule that corresponds to this new string as a neighbour solution is presented in Figure 5, and the makespan increases to 9 clock cycles.

One may notice that the warp pseudo-precedence string is a much more low-level representation (compared with the corresponding schedule), but because of the fact that it does not bind the warps to particular clock cycles, we are free to make permutations of the warps in the string subject to all the logic of the kernel, capacity, precedence and work-conserving constraints, even though these are not explicitly specified in terms of the data structure itself.

Note that even different unique permutations do not necessarily specify unique solutions. For example, the warp pseudo-precedence string "1 2 1 3 2 3 1 2 3 4 4 4" still corresponds to the schedule in Figure 5 (built according to the different string "1 1 2 2 3 3 1 2 3 4 4 4").

V. The metaheuristic

As shown in Section IV, we present finding the maximum makespan as a combinatorial optimization problem where a solution is sought over a discrete search-space of warp pseudo-precedence strings. Considering even a relatively moderate length for the kernel instruction string (I) and the number of warps (W), the brute-force search over $\frac{(W \cdot I)!}{(I!)^W \cdot W!}$ permutations (Section 4 in the report [5]) would not be computationally tractable. Consequently, we apply computational methods that iteratively search for a "better" solution according to a given strategy.

Among many different metaheuristics that are widely used in various scientific and application domains we decided in favour of simulated annealing by Kirkpatrick et al. [12]. This metaheuristic is very popular for tackling combinatorial problems and in our particular case the warp pseudo-precedence string as a solution is amenable for applying simulated annealing to it. Inspired by the annealing technique in metallurgy, simulated annealing attempts to replace the current solution of the problem with another candidate solution (often randomly obtained) at each its iteration. A candidate solution that improves on the current one is always accepted. However, occasionally, the algorithm will also accept a "worse" candidate solution with a probability which depends on the value of probability function. This function takes as parameters a variable T (also called as "the temperature") and the difference of the utilities of the current solution and the candidate solution. Higher temperatures and lower reduction in utility makes it more likely that such a candidate solution will be chosen. Occasionally accepting "worse" solutions helps avoid the pitfall of getting stuck at a local optimum of the optimization problem. With the number of iterations, T is decreased according to a given "annealing schedule".

Let $iter_{max}$ denote the (user-defined) maximum number of iterations for the annealing and let the variable $iter$ hold the index of the current iteration. Before the first iteration the temperature T is set to T_0 and is decreased after every iteration according to the following annealing schedule:

$$T = T_0 \cdot \left(1 - \frac{iter}{iter_{max}}\right)$$

The lower the temperature is set, the more "greedy" (in its preference for better solutions) the metaheuristic becomes. This principle is specified in the definition of the probability function which, besides T, also depends on the makespans of the current (m) and the candidate solution ($m^{cand.}$):

$$P(m, m^{cand.}, T) = \begin{cases} 1 & \text{if } m^{cand.} \geq m; \\ \min(1, \frac{T}{m - m^{cand.}}) & \text{otherwise.} \end{cases} \quad (3)$$

Note how the probability of accepting a solution with a smaller makespan decreases as $(m - m^{cand.})$ increases.

VI. Implementation optimization

Although any "randomly" shuffled string consisting of I instances of each warp identifier could serve as an initial solution, providing a "good" initial solution to the metaheuristic may considerably speed up the convergence towards a good estimate of the makespan. Hence, although our technique is parallelizable over an arbitrary degree of processors (which would help with convergence speed), in the report [5] we present some "templates" (according to our empirical observation) for generating initial solutions with long makespan. In brief, the "round-robin" template corresponds to the string $\underbrace{1, 2, \ldots, W, 1, 2, \ldots, W, \ldots, 1, 2, \ldots, W}_{\text{I times}}$, whereas the "fixed priority" template corresponds to $\underbrace{1, 1, \ldots 1}_{\text{I times}}, \underbrace{2, 2, \ldots 2}_{\text{I times}}, \ldots \underbrace{W, W, \ldots W}_{\text{I times}}$ The template "most pending warp executes first" is constructed according to a more complex heuristic described in the report [5]. When running the metaheuristic on a multi-processor machine (with one thread per processor), we recommend using the initial solutions based on these templates on some processors and random warp pseudo-precedence strings on the rest.

For each new candidate solution considered, the metaheuristic needs to create a corresponding schedule from the new warp pseudo-precedence string under consideration using the algorithm of Figure 4, so that the corresponding makespan can be calculated. Doing so from scratch could be an option, but would be inefficient, in the sense that, if each neighbour solution was obtained just by a single permutation (or a few) of the warp pseudo-precedence string, then surely the two schedules would be similar and, in principle, there should exist a faster way, of deriving the one from the other by doing just the part of the computation reflecting the differences of the two pseudo-precedence strings. Over a large number of iterations the time saved would be significant as the convergence to a good estimate of the makespan would be sped up.

Therefore, we introduce the warp cycle string $warpCycleStr$ – an integer string of the same length as the warp pseudo-precedence string $warpPrecStr$. Element $warpCycleStr[w]$ holds the index of the clock cycle in which the warp with the identifier $warpPrecStr[w]$ is scheduled. As an example, for the warp pseudo-precedence string (2) the warp cycle string is the following "1 2 2 3 3 4 4 5 5 6 7 8" and it can be easily verified using the schedule in Figure 2.

The $warpCycleStr$ itself is a "compact" way of storing a schedule (instead, e.g. of sparse two-dimensional arrays). If the first index where the new $warpPrecStr$ differs from the previous one is z, then, from elements $warpCycleStr[1]$ to $warpCycleStr[z-1]$ we can obtain the "common" part of the schedule. It then suffices to assign new values for elements $warpCycleStr[z]$ onwards, considering the rest of the new pseudo-precedence string (i.e. from $warpPrecStr[z]$ onwards).

VII. CASE STUDIES

A. Overview

The technique presented in previous work [4] for finding the exact worst-case makespan can also be used to find in tractable time an upper bound on the worst-case makespan by conservatively combining results of subwarp makespans. However, even that sometimes takes long to compute. The technique in this paper, by comparison, may be used to derive a tight lower bound on what the output of ILP-based approach [4] would have been. Since the previous work [4] is potentially pessimistic, this means that the technique in this paper provides an estimate that, at best, is conservative – and at worst, it is a slight underestimation of the true worst-case makespan. We implemented our technique as a multithreaded module.

Parameters for the problem instance under consideration can be categorized as (i) program-related (the number of warps; the kernel instruction string), (ii) hardware-related (the number of computational units of each type; the warp size) and (iii) metaheuristic-related (the initial temperature T_0, the maximum number of iterations $iter_{max}$ and an integer flag specifying the kind of the initial solution – i.e. whether it is random or obtained according to one of the patterns presented in Section 6 of the report [5]. These parameters serve as input to each thread on the respective processor, which then starts to iterate among candidate solutions, in parallel with and independently of other threads on other processors. The estimate, at any instant, is obtained as the greatest reported makespan so far, over all threads.

B. The benchmark

For our experiments, we chose a kernel instruction string derived from a real application that could be run as many parallel GPU threads: Voronoi diagrams which are used e.g. for solving proximity problems in computational geometry or localization in wireless sensor networks. We consider a massively parallel Voronoi diagram rendering application presented in Section 8B of the report [5].

Our "port" of that program to assembly for NVIDIA's Parallel Thread Execution (PTX) virtual machine [21] is shown in Figure 6. Every line consists of an assembly statement, comments that "map" that statement to the corresponding code from the original higher-level program illustrated in Figure 10 of [5] and a character for the type of hardware unit assumed to perform the corresponding assembly instruction. We tag instructions executed on CUDA core with a "C" and instructions for a load/store unit with an "L". The resulting kernel instruction string corresponding to the branchless code, from the start of the program until the end of the first iteration of the inner loop in Figure 6, was used in our experiments.

```
        mov.u32   $r0,   N_addr;          // N                                  L
        mov.f32   $f1,   x_addr;          // x                                  L
        mov.f32   $f2,   y_addr;          // y                                  L
        mov.f32   $f3,   x1_addr          // x1                                 L
        mov.f32   $f4,   y1_addr;         // y1                                 L
        sub.f32   $f5,   $f1,   $f3;      // (x-x1)                             C
        mul.f32   $f6,   $f5,   $f5;      // (x-x1)^2                           C
        sub.f32   $f7,   $f2,   $f4;      // (y-y1)                             C
        fma.f32   $f8,   $f7,   $f7,  $f6;// (x-x1)^2+(y-y1)^2                   C
        mov.f32   $f0,   $f8;             // md = (x-x1)^2+(y-y1)^2             C
        mov.u32   $r1,   1;               // int mdp = 1;                       C
        mov.u32   $r2,   2;               // for (int i=2; i<=N; i++)           C
Loop:   setp.gt.u32 p,   $r2,   $r0;      // for (int i=2; i<=N; i++)           C
@p      bra Done;                         // for (int i=2; i<=N; i++)           C
        mov.f32   $f3,   xi_addr;         // xi                                 L
        mov.f32   $f4,   yi_addr;         // yi                                 L
        sub.f32   $f5,   $f1,   $f3;      // (x-xi)                             C
        mul.f32   $f6,   $f5,   $f5;      // (x-xi)^2                           C
        sub.f32   $f7,   $f2,   $f4;      // (y-yi)                             C
        fma.f32   $f8,   $f7,   $f7,  $f6;// (x-xi)^2+(y-yi)^2                   C
        setp.ge.f32 q,   $f8,   $f0;      // if ( (x-xi)^2+(y-yi)^2 <md)        C
@q      bra If;                                                                C
        mov.f32   $f0,   $f8;             // md = (x-xi)^2+(y-yi)^2 ;           C
        mov.u32   $r1,   $r2;             // mdp = i;                           C
If:                                       // if ( (x-xi)^2+(y-yi)^2 <md)
        add.u32   $r2,   $r2,   1;        // for (int i=2; i<=N; i++)           C
        bra Loop;                                                              C
Done:
```

Fig. 6. PTX program for visualizing Voronoi diagrams.

C. Experimental results

The metaheuristic approach described outputs a lower bound on the worst-case makespan for the problem instance in consideration under the simplifying assumptions discussed earlier. These assumptions were all pessimistic except for the assumption that all load/stores are single-cycle. Conversely, the ILP-based approach [4] outputs an upper bound for the worst-case makespan under the same assumptions. Therefore, we sought to investigate the "quality" of the solutions output by the metaheuristic by comparing its output with that of the ILP-based approach.

As benchmark, we used the Voronoi kernel instruction string introduced earlier:

$$\underbrace{LLLLL}_{5\ Ls}\underbrace{CCCCCCCCC}_{9\ Cs}\underbrace{LL}_{2\ Ls}\underbrace{CCCCCCCCC}_{9\ Cs}$$

We used parameters $\{\sigma_C = 4,\ \sigma_L = 1\}$ (intended to model NVIDIA Kepler, under the pessimistic assumption that only one instruction dispatch unit per warp scheduler is used) and for $W = 16$ warps. We ran 8 instances (Java threads) of the metaheuristic (2 with the "round-robin" initial solution; 2 with "fixed-priority"; 2 with "most pending warp executes first"; 2 random) with initial temperature $T_0 = 0.3$ for $2 \cdot 10^6$ iterations each on a Pentium Dual-core E5400 (2.7 GHz). These runs were performed sequentially, not in parallel. However, by logging every reported improvement to the current estimate along with timestamps, in seconds since the beginning, we were able to retroactively "simulate" the behaviour one would get by running the instances of the metaheuristic in parallel, since their executions would be independent anyway. The reported estimates of the individual Java threads are plotted in Figure 7, with the horizontal axis denoting the time since launch. The composite reported estimate, obtained as the maximum over all graphs, at any time instant (i.e. as the "envelope" of all graphs), converged to 160 at the end of the experiment.

By comparison, the upper bound on the worst-case makespan obtained via the ILP-based approach for 16 warps was 176 clock cycles and took 58 hours to compute, on the same machine. It was derived by pessimistically extrapolating from the respective exact worst-case estimate for 4 warps, which was the most that could tractably be computed. This means that the estimate by the metaheuristic was just 9.1% lower than the one by the ILP-based approach. We interpret this as evidence that both approaches provide relatively tight lower/upper bounds respectively for the worst-case makespan,

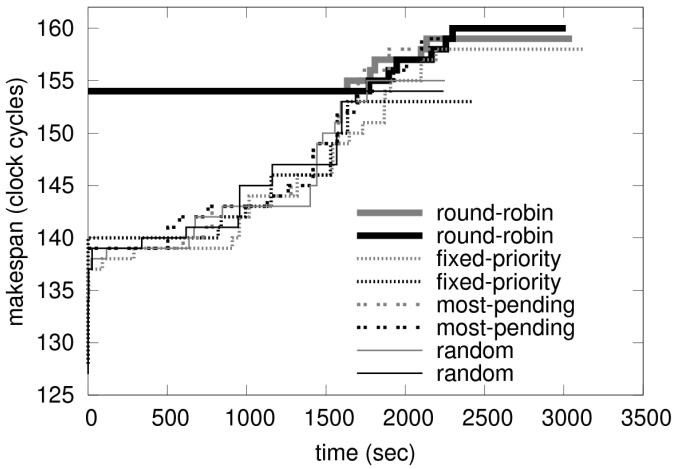

Fig. 7. Convergence of the estimates of the worst-case makespan over time, for 8 instances of the metaheuristic, with different initial solutions.

subject to our assumptions. However, the metaheuristic provides its estimates orders of magnitude faster.

Additional observations from this small-scale experiment are that, even the "round-robin" initial solution can serve as a quick/rough estimate for the worst-case makespan (even *before* running the meta-heuristic). This is also in accordance with our experience by experimenting with other kernel instructions strings and problem instances in general. However, even when the metaheuristic is launched with random initial solutions, it converges fast towards better estimates, comparable to those obtained when using the "round-robin" initial solution. The graphs also serve, to an extent, to highlight the relative speedup that can be achieved in the convergence to a good estimate, by running (and tracking) multiple independent instances of the metaheuristic in parallel.

VIII. Conclusion

This paper presented a tractable technique to obtain an estimate of the worst-case makespan of a set of identical GPU threads running on a single streaming multiprocessor, subject to some simplifying assumptions. This technique is based on the metaheuristic of simulated annealing and is readily parallelizable, for even faster convergence. The result is very close to the pessimistic estimate obtainable using a much more computationally complex ILP-based technique. Therefore, the estimate output by this new technique is, in the most unfavourable circumstance, a slight underestimation of the actual worst case. As such, the target of the approach is soft real-time systems, wherein a very rare missed deadline does not matter. As a next step, for additional confidence, and even though the degree of latency hiding makes this less of an issue, we will aim to relax the current optimistic modelling of the memory subsystem.

References

[1] Advanced Micro Devices, Inc. ATI Stream Computing Programming Guide. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_CAL_Programming_Guide_v2.0.pdf, 2010.

[2] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.

[3] N. C. Audsley and K. Bletsas. Realistic analysis of limited parallel software / hardware implementations. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS2004)*, Toronto, Canada, may 2004. IEEE Computer Society, IEEE.

[4] K. Berezovskyi, K. Bletsas, and B. Andersson. Makespan computation for GPU threads running on a single streaming multiprocessor. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*.

[5] K. Berezovskyi, K. Bletsas, and S. M. Petters. Fast Makespan Estimation for GPU Threads on a Single Streaming Multiprocessor. Technical Report CISTER-TR-130406, CISTER/INESC-TEC, ISEP Research Center, Polytechnic Institute of Porto, Available at http://www.cister.isep.ipp.pt/people/Kostiantyn%2BBerezovskyi/ publications/, April 2013.

[6] K. Bletsas and N. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*, pages 525 – 531, aug. 2005.

[7] P. Blixt, M. Kobetski, and A. Höglund. Eye-tracking using a gpu. Patent, 2011.

[8] CHRONOS VISION GmbH. Deployment in clinical and research labs. Product brief – http://www.chronos-vision.de/en/eye-tracking-development.html, 2012.

[9] G. Elliott, B. Ward, and J. Anderson. Gpusync: Architecture-aware management of gpus for predictable multi-gpu real-time systems. In submission.

[10] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[11] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, 2012.

[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[13] Konstantinos Bletsas. Worst-case and Best-case Timing Analysis for Real-time Embedded Systems with Limited Parallelism. PhD thesis, 2007.

[14] B. Lisper. Towards parallel programming models for predictability. In T. Vardanega, editor, *Proc. 12th International Workshop on Worst-Case Execution-Time Analysis (WCET'12)*. Schloss Dagstuhl, July 2012.

[15] R. Mangharam and A. A. Saba. Anytime algorithms for gpu architectures. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, December 2011.

[16] Mirametrix Inc. S2 Eye Tracker. Product brief – http://mirametrix.com/products/eye-tracker/, 2011.

[17] J. B. Mulligan. A gpu-accelerated software eye tracking system. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA '12, pages 265–268, New York, NY, USA, 2012. ACM.

[18] 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.

[19] NVIDIA Corp. NVIDIA GeForce GTX 680. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf, 2012.

[20] NVIDIA Corp. NVIDIA's next generation CUDA compute architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[21] NVIDIA Corporation. NVIDIA CUDA C programming guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2012.

[22] 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.

[23] Tobii Technology. Tobii eyeasteroids the world's first eye-controlled arcade game. Product brief – http://www.tobii.com/Global/FutureMarkets/EyeAsteroids/Documents/Tobii_Leaflet_EyeAsteroids_02112011_WEB.pdf, 2011.

[24] Tobii Technology. Advanced driver assistance with Tobii eye tracking. Product brief – http://www.tobii.com/gaze-interaction/global/Gaze-Interaction-fields-of-use/advanced-driver-assistance/, 2012.

[25] 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.