A Simple BSP-based Model to Predict Execution Time in GPU Applications

Marcos Amarís, Daniel Cordeiro, Alfredo Goldman
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, Brazil
{*amaris, danielc, gold*}*@ime.usp.br*

Raphael Y. de Camargo
Departament of Computer Sciences
Federal University of ABC
Santo André, Brazil
raphael.camargo@ufabc.edu.br

***Abstract*—Models are useful to represent abstractions of software and hardware processes. The Bulk Synchronous Parallel (BSP) is a bridging model for parallel computation that allows algorithmic analysis of programs on parallel computers using performance modeling. The main idea of BSP model is the treatment of communication and computation as abstractions of a parallel system. Meanwhile, the use of GPU devices are becoming more widespread and they are currently capable of performing efficient parallel computation for applications that can be decomposed on thousands of simple threads. However, few models for predicting application execution time on GPUs have been proposed.**

In this work we present a simple and intuitive BSP-based model for predicting the CUDA application execution times on GPUs. The model is based on the number of computations and memory accesses of the GPU, with additional information on cache usage obtained from profiling. Scalability, divergence, effect of optimizations and differences of architectures are adjusted by a single parameter.

We evaluated our model using two applications and six different boards. We showed by using profile information for a single board, that the model is general enough to predict the execution time of an application with different input sizes and on different boards with the same architecture. Our model predictions were within 0.8 to 1.2 times the measured execution times, which are reasonable for such a simple model. These results indicate that the model is good enough to generalize the predictions for different problem sizes and GPU configurations.

***Keywords*-BSP model, Performance Prediction, GPGPU, Kepler Architecture, CUDA.**

I. Introduction

Graphics Processing Units (GPUs) are specialized processing units that were initially conceived with the purpose of accelerating vector operations, such as graphics rendering. GPUs are general purpose parallel processing units with accessible programming interfaces, including standard languages such as C and Python. In particular, the Compute Unified Device Architecture (CUDA) is a parallel computing platform that facilitates the development on any GPU enabled system [1]. CUDA was introduced by NVIDIA in 2006 for their GPU hardware line.

Parallel computing models have been an active research topic since the development of modern computers [2], [3], [4]; their main goal is to provide a standard way of describing and evaluating the performance of parallel applications. For the success of a parallel computing model, it is paramount to also consider the characteristics of the underlying architecture of the hardware being used.

One of the most well-established models for parallel computing is the Bulk Synchronous Parallel (BSP), first introduced by Valiant in 1990 [5]. Its main goal was to provide a bridging model that can represent different architectures sufficiently well, without considering all the hardware details. The BSP model bridges the essential characteristics of different kinds of machines as a combination of three attributes:

- a set of virtual processors, each associated to a local memory;
- a router, that delivers the messages in a point-to-point manner;
- a synchronization mechanism for all or for a subset of processors.

The computation is organized in a sequence of *supersteps*, each one divided into three successive—logically disjointed—phases. On the first phase, all processors use their local data to perform local sequential computations in parallel (i.e., there is no communication among the processors.) The second phase is a communication phase, where all nodes exchange data performing personalized all-to-all communication. The last phase consists of a global synchronization barrier, that guarantees that all messages were delivered and all processors are ready to start the next superstep. Figure 2 depicts the phases of a BSP application. On the BSP model there is no restriction on sending messages, but all of them should be received by the synchronization barrier. According to the execution model, the first and second phase may occur simultaneously.

In this work we propose a new refinement of the BSP model for applications developed for GPUs. Similarly to the BSP model, our model is mainly based on the number of computational and communication steps used by the application. These values are multiplied by parameters that describe the number of cores, threads and the clock rate of the processors. Differently from the BSP model, we did not include the synchronization step of the BSP model, since global synchronizations in GPUs occur only at the end of the kernel and we consider only the execution of a single kernel.

Application optimizations, such as divergence, shared bank conflicts, and coalesced global memory accesses can have an important impact on the performance of GPU applications. We model these effects by adjusting a single parameter λ, which can be obtained empirically by executing one instance of the application on a single GPU model and with a single input size. The same λ value would them be used to predict the application execution time for other GPU models and input sizes.

The predictability of this model is evaluated with the classical use-case of matrix multiplication application (with 4 different types of performance optimizations) and with an application to solve the maximum subarray problem [6]. We evaluated our model using 6 different GPUs models, comparing the predicted execution times with the actual measured times. We verified that we could use the same λ value for all GPU models and for a large range of input sizes, showing that this parameter could be general enough to capture the optimization effects in the evaluated applications.

The structure of this document is as follows. Section II reviews the GPU architectures and the programming model of CUDA, while Section III describes other works about parallel models in GPUs. In Section IV we present the model proposed in this research. Section V explains the different use-cases used to evaluate our new model and Section VI shows our experimental results. Finally, Section VII concludes this work and presents the planned future works.

II. GPU Architecture and CUDA

The most widely used GPUs are those produced by NVIDIA, together with the CUDA platform. The architecture of these GPUs is built with a set of Streaming Multiprocessors (SMs) each containing several cores called Scalar Processors (SPs), a set of Special Function Units (SFUs) and a number of load/store units. The multiprocessors execute asynchronously, in parallel. The SM schedules threads in groups of 32 parallel threads called warps, which can use load/store units concurrently, allowing simultaneous reads from memory for these threads.

GPUs have a hierarchical memory, with a large, high latency off-chip global memory and a small, low latency on-chip memories, such as the shared memory and registers. Each GPU has its own DRAM, referred to as global memory, which can be accessed from any multiprocessor. The GPU also has a noncoherent caches. Each multiprocessor has its own cache L1. Data in the shared memory can be accessed by multiple hardware thread contexts on the same multiprocessor.

Each thread has to access these different levels of memory, and their efficient usage are essential for optimizing application execution. More recent boards have L1 and L2 cache memories to reduce the latencies of global memory accesses.

The GPU architecture has evolved in recent years, with Tesla, Fermi, Kepler, and Maxwell architectures. Their main architectural differences are in the configuration of SMs, such as number of cores, registers, SFU and load/store units, on-chip memory size and local cache. There are also changes in processor clock frequency, memory bandwidth, and some specific features, such as unified memory spaces and dynamic kernel launches, features summarized in the Compute Capability of the GPU. A function launched in a GPU with CUDA programming model is named kernel.

Tesla, Fermi and Kepler architectures have SFUs to execute transcendental instructions such as *sin*, *cosine* and *square root*, with each SFU executing one instruction per clock. Tesla architectures implemented fused multiply-add (FMA) for double precision. Fermi architectures implemented the IEEE 754-2008 floating-point standard [7] for both single and double precision arithmetic which performs multiplication and addition with a single rounding step.

Similarly to Tesla, Fermi and Kepler architectures support a unified memory request path [8]. In Tesla architectures, global memory access by threads of a half warp (16) can be coalesced to one transaction for words of sizes 8-bit, 16-bit, 32 bit, 64-bit and 128-bit. On Fermi and Kepler architectures, coalesced accesses can be done by all threads of a warp.

The global memory has a latency of about 400 or 600 cycles per clock [9], [10], which has a latency of one cycle. To improve memory access efficiency, Fermi and Kepler provide a low-latency on-chip L1 cache, with an access latency equal to the shared memory. A L2 off-chip cache is also present, with a latency higher than L1, but lower than the global memory. L1 and shared memory are accessed only by threads from the same SM, while global memory and cache L2 are accessed by every thread. Figure 1 shows the hierarchy memory accessed by any thread executed in a Fermi and Kepler architecture, this figure was adapted from [11].

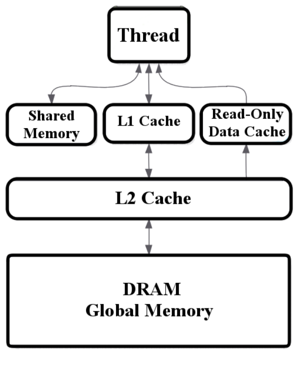

Figure 1. Memory hierarchy of threads in a kernel executed in Kepler architectures

A. *Compute Unified Device Architecture*

The Compute Unified Device Architecture (CUDA) is a high-level platform for developing GPU applications. It extends the C language and provides a compiler that translates it into a pseudo-assembly PTX (Parallel Thread Execution) code, which is executed in NVIDA GPUs. CUDA applications are organized in kernels, which are functions executed on GPUs. GPUs execute kernels asynchronously and, depending on the Compute Capability, several kernels can be executed concurrently.

Threads blocks are assigned to SMs, which can concurrently execute groups of threads called warps. Threads from the same warp must execute the same instructions. When the code contains conditional flows, threads from the same warp may have to execute different instructions, what is performed sequentially, in an effect called divergence. Divergences can affect the efficiency of applications.

The performance of a kernel execution in a GPU depends largely on the optimization of access to data in the memory hierarchy Threads within a block can cooperate by sharing data through the shared memory. Shared memory is on chip in each multiprocessor and has a very small latency.

This memory is divided into memory banks and different banks can be accessed concurrently, adjacent 4-byte words are stored in adjacent banks. But only one thread can access each bank in the same clock cycle, if multiple threads try to access different locations in the same bank, all memory accesses will be serialized, this is known as bank conflicts. Consequently, applications should be organized so that threads from the same warp access data from different memory banks.

The bandwidth of the global memory can be largely improved by combining the load/store requests from different threads of a single warp in a single memory request, in a process called coalescing [12]. The coalescing occurs when the threads access contiguous global memory addresses, which permits usage of the multiple load/store units available per SM.

III. Related Work

The BSP model has been widely used on different applications contexts. HPC practitioners have been using the BSP model to design algorithms and software that can run on any standard architecture with guaranteed performance [13], [14], [15]. Consider a BSP program that runs on S supersteps using p processors simultaneously with clock rate (speed) R. Let g (the *gap*) be the bandwidth of the network and L the latency—i.e., the minimum duration of a superstep—which reflects not only the latency of the network, but also the overhead of the synchronization step.

The cost to execute the i-th superstep is then given by:

$$w_i + gh_i + L \tag{1}$$

where w_i is the maximum amount of local computations executed, and h_i is the largest number of packets sent or received by any processor during the superstep. If $W = \sum_{i=1}^{S} w_i$ is the sum of the maximum work executed on all supersteps and $H = \sum_{i=1}^{S} h_i$ the sum of the maximum number of messages exchanged in each superstep, then the total execution time of the application is given by:

$$T = W + gH + LS \tag{2}$$

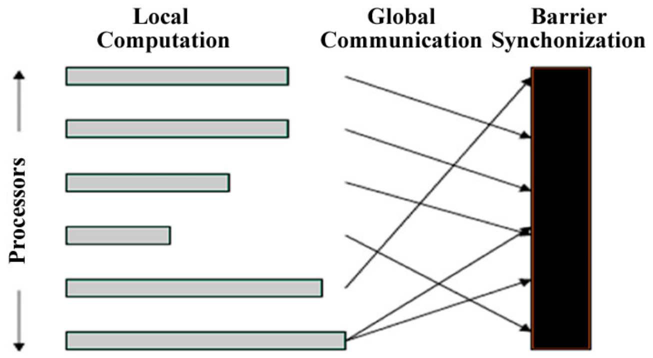

Figure 2. Superstep in a Bulk Synchronous Parallel Model.

It is common to present the parameters of the BSP model as a tuple (w, g, h, L).

There are other parameterized parallel models [3], [4], almost all of them using or extending the core of the BSP model. Dehne et al. [16] have studied the problem of designing scalable parallel geometric algorithms, and he has introduced the Coarse Grained Multicomputer model (CGM), where a set of p processors are constrained to use $O(N/p)$ local memory, where N is the input size of the problem.

To ease the development and analysis of parallel programs, the BSP programming model was implemented as API libraries [17], and recently enhanced to simplify programming on GPU architectures [18]. These developments help to create scientific applications in massively parallel environments computing in an easier and better way.

Recently, Valiant proposed an adaptation of the BSP model over multicore architectures; he called this model Multi-BSP [19]. Multi-BSP incorporates the memory size as an additional parameter. Multi-BSP model is a multi-level model that has explicit parameters for the number of processors, memory cache sizes, communication costs, and synchronization costs. It recognizes the physical characteristics of multiple memory and cache levels both within single chips as well as in multi-chips architecture.

BSP and Multi-BSP propose a bridge between the development and analysis of algorithms over distributed and parallel systems, but they make emphasis in machines with multicore processors and not on GPUs.

Kothapalli et al. [20] have presented a combination of known models with small extensions. The models they have used are: BSP model, PRAM model by Fortune and Wylie [21] and the QRQW model by Gibbons [2].

The authors abstract the GPU computational model by considering the pipeline characteristic of the application in GPU architectures. But they do not consider the effects of divergence, which can have a significant impact in the efficiency of GPU applications and developers need a strong understanding of parallel applications in CUDA and its different optimizations.

Hong and Kim [22] have proposed and evaluated a memory and parallelism-aware analytic model to estimate execution time of massively parallel application in GPUs. The key idea is to find a metric which they have called MWP (Memory Warp Parallelism) and CWP (Compute Warp Parallelism). The analytic model provides good performance predictions, however, this model requires a deep analysis and understanding by third-party developers of parallel applications in CUDA. They have introduced the metrics MWP and CWP, MWP is related to how much memory parallelism in the application and CWP is related to the program characteristics. CWP is used to decide whether performance is dominated by computation or communication.

Zhang et al. [23] have presented a quantitative performance analysis model, based on micro-benchmarks for NVIDIA GeForce 200-series GPUs. They have developed a throughput model for three components of GPU execution time: the instruction pipeline, shared memory access, and global memory access. The model is based on a native GPU instruction set instead of the intermediate PTX assembly language or a high-level language. Our model uses a high-level bridging model for parallel computation and is focused on computation and communication processes for any GPU application. This encourage developers to use better optimizations in communication and computation.

Kirtzic has proposed the the Parallel GPU model (PGM) [13]. PGM is an adaptation of the models PRAM, BSP and parallel phase model. This model offers a general design and a fine-grained approach. PGM is not a model to predict time; on the contrary, it is a parallel algorithm development model. The model can result in significant increase in performance when algorithms are designed based on their principles. This parallel GPU method allows parallel GPU algorithm designers, ranging from the novice to the expert, to design and implement optimal algorithms that take advantage of GPU architectures. To achieve optimal for particular algorithms, knowledge about the architecture of specific GPUs is required.

Kerr et al. developed a methodology for the systematic construction of performance models of heterogeneous processors [24]. This methodology is comprised of experimental data acquisition and database construction, a series of data analysis passes over the database, and model selection and construction. They developed a framework, named Eiger, that implements their methodology. Another framework to construct performance models was presented by Spafford and Vetter [25]. They used a domain specific language to develop analytical performance models for the three dimensional Fast Fourier Transform (3D FFT).

GPU applications can hide computation and communication latency by executing many parallel threads in an interleaved mode. When the instruction pipeline is fully saturated, the performance of the application run close to the peak performance of the GPU. In contrast, when the pipeline is under-utilized, the situation lies about the performance of the application [23], [26].

Based on recent studies about the impact of small cache hierarchy on performance, it is critical to model the GPU caches [27]. Also, the models must carefully consider the effects of memory access divergence, otherwise it can significantly decrease the fidelity of the model [28], [29].

Our model is based solely in the BSP model. Scalability, optimization effects and differences between architectures are all adjusted by a single parameter λ. Profiling techniques were used to confirm information about the behavior of applications and to establish parameters about computation and communication processes in GPU applications. Our proposed model offers a simple analytical model can be used to predict performance of GPU applications. This model allowed an easy parametrization, well-suited for any GPU applications in practice.

IV. Proposed Model

We propose a new simple performance prediction method for GPU applications based on the BSP model. Similarly to BSP, our model considers communication and computation as an abstraction of a parallel system, and takes into account the main physical properties and optimizations of GPU architectures. The performance prediction is based on the cost of communication and computation, which are determined independently.

The model focus on the execution time prediction of a single kernel function. The execution time is split between computation and data transferring to and from global and shared memories.

$$T_k = \frac{t \cdot (Comp + Comm_{GM} + Comm_{SM})}{R \cdot P \cdot \lambda} \tag{3}$$

In Equation 3, T_k is the approximated execution time of a kernel function with t threads. It sums the computational cost ($Comp$) with the communication cost of global memory ($Comm_{GM}$) and shared memory ($Comm_{SM}$) accesses, performed by each thread. This cost is multiplied by the number of threads t and divided by the clock rate R times the number of cores P available in the GPU. The parameter λ is used to model the effects of application optimizations, such as divergence, shared bank conflicts and coalesced global memory accesses.

The computational time used by each thread in a kernel is denoted by $Comp$. It is determined by the number of cycles that each thread spends in its computation. FMA operations can be included in $Comp$ by reading the source code of the kernel and verifying this possibility with profiling tools.

Communication is evaluated at two levels: global and shared memory. The execution time for communication in global and shared memory per thread are given by $Comm_{GM}$ and $Comm_{SM}$, respectively. These are defined as the sum of load and write transactions over the global memory and shared memory. This information can be extracted directly from the source code.

Additionally, to account the effects of cache memories on recent GPU architectures, the number of L1 and L2 cache hits are subtracted from the number of loads over the global memory. We have used metrics and events to confirm information about the number of L1 and L2 cache hits. Their contribution to the execution time is calculated separately, multiplying them by their latency times [10][30]. This model allows an easy parametrization, well-suited for any GPU applications in practice. For simplification, we do not consider constant and texture memories nor differences between the latency of load and store transactions. $Comm_{SM}$ and $Comm_{GM}$ are defined as:

$$Comm_{SM} = (ld_0 + st_0) \cdot g_{SM} \tag{4}$$

$$\begin{aligned} Comm_{GM} \quad = \quad & (ld_1 + st_1 - L1 - L2) \cdot g_{GM} + \\ & L1 \cdot g_{L1} + L2 \cdot g_{L2} \end{aligned} \tag{5}$$

g_{SM}, g_{GM}, g_{L1} and g_{L2} represent the latency in communication over shared, global, L1 cache and L2 cache memory, respectively. Some typical values are 5 cycles for g_{SM} and g_{L1}, 500 cycles for g_{GM} [9], and 250 cycles for g_{L2}.

ld_0 and st_0 represent the total number of load and stores performed by all threads in the shared memory, and ld_1 and st_1 represent the loads and stores for global memory. The number of loads and stores to global and shared memory are determined by analyzing the CUDA source code. $L1$ and $L2$ are determined executing an application execution profile, resulting in a number between 0 and 1, which is multiplied by the size of the problem N.

Application optimizations, such as divergence, shared bank conflicts and coalesced global memory accesses, are important to define the application performance [28], [31]. We consider the effects of those optimizations using the λ factor. It is estimated as the ratio between the predicted execution time of the application with the actual measured execution time. The λ factor is important since it permits the adjustment of application performance with the implemented CUDA optimizations and GPU architectures. Finally, intra-block synchronization is not computed, since it does not affect processing time [9], [32].

```
 1 __global__ void matMul(float* Pd, float* Md,
 2                         float* Nd, int N) {
 3   float Pvalue = 0.0;
 4   int j = blockIdx.x * tWidth + threadIdx.x;
 5   int i = blockIdx.y * tWidth + threadIdx.y;
 6
 7   for (int k = 0; k < N; ++k)
 8     Pvalue += Md[j * N + k] * Nd[k * N + i];
 9
10   Pd[j * N + i] = Pvalue;
11 }
```

Figure 3. Kernel in CUDA of matrix multiplication only with global memory and no-coalesced accesses.

Consequently, except for the value of λ and effects on caches L1 and L2, all other parameters are constants. The effect of usage of caches L1 and L2 must be confirmed by profiling. λ performs the adjustment of application performance with the implemented CUDA optimizations. Once defined for the application, the same value should work for other GPU architectures and input sizes of the application.

V. USE CASES

We have applied the model with two applications, *Matrix Multiplication* and *Maximum Subarray Problem* [6], developed in CUDA using the single-precision format. Both applications use a single kernel. We have chosen a average GPU (GTX-680, see table I) to find the parameter $lambda$ in our simulations of the model. We verified that we could use the same λ value for all GPU models and a large range input sizes.

During our evaluation, all applications were executed using the CUDA profile tool *nvprof*. Each experiment is presented as the average of ten executions, with a confidence interval of 95%.

A. Matrix Multiplication

We have used four different optimization techniques regarding the use of the available memories for the matrix multiplication application: (#1) global memory only; (#2) global memory with coalesced accesses; (#3) shared memory without coalesced accesses to global memory; and (#4) shared memory with coalesced accesses to global memory.

The running time of a matrix multiplication for two matrices of size $N \times N$ is proportional to $O(N^3)$ in the sequential algorithm and to $O(N)$ in CUDA, using N^2 threads. We have adopted `tWidth`^{2} threads per block and defined the number of blocks to be square of `(N + tWidth-1)/tWidth`, dynamically devised from the size of the problem (N), `tWidth` is equal in all the optimizations.

Figure 3 shows the CUDA source code in the first optimization mode. Non-coalesced accesses occurs because of irregular references to data in global memory. Lines 7 and 8 of the algorithm show that each thread performs N single precision fused multiply-add (FMA) arithmetic operations and N reads from global memory for each matrix. A single write operation is performed in line 10. We do not consider the accesses to the registers. All communication in this mode is performed over global memory; shared memory is not used.

$Comp$ is determined by the number of multiplications and/or operations computed by a thread. In this case, each thread performs N FMA single precision operations. IEEE 754-2008 floating-point standard [7] states that those operations needs a single rounding step or cycle. The value of $Comp$ is the same for the all four optimizations modes, since they differ only in the memory access patterns.

The number of accesses to global and shared memories are constants, with ld_0 and ld_1 being N and st_0 and st_1 being 1 for all optimization modes. The number of cache hits varies across executions for different sizes of the problem because of changes on the rate of coalesced accesses.

With those values, we can compute—using equations 5 and 4—the values of $Comp$, $Comm_{GM}$, and $Comm_{SM}$, that are then multiplied by the number of threads t in the kernel.

The optimizations actually affect only the performance of the communication between threads. For the optimization (#1), we have found empirically $\lambda = 4.35$. As explained above, $\lambda = 1$ in the first execution and it is obtained by the ratio of the predicted execution time of the application with the actual measured execution time. This means that the profiling of the application and the source code analysis gave a preliminary version of the model that were about four times smaller than the actual execution time.

This model allowed an easy parametrization, well-suited for any GPU applications in practice. With a few knowledge of other GPUs, the model extracted of this application can represent good approximations of the time execution of this application.

For optimization (#2), we changed the data access pattern, to permit coalesced access to data in global memory. In Figure 3 line 8 is changed to

```
Pvalue += Md[j * Width + k] * Nd[k * Width + i];
```

and line 10 is changed to

```
Pd[j * Width + i] = Pvalue;
```

in this case we have have obtained $\lambda = 23$.

Optimization (#3) uses shared memory to load data from global memory and to process them with a lower latency of communication. Similarly to the previously examined optimizations, optimization (#4) is obtained by changing the coalesced memory access from the source code of optimization (#3). The source code for optimization (#4) is shown in Figure 4.

The external loop in lines 11–19 transfers data from global to the shared memory. The internal loop, in lines 16 and 17, performs the actual matrix multiplication computation.

```
 1  __global__ void matMul(float* Pd, float* Md,
 2                                   float* Nd, int N){
 3    __shared__ float Mds[tWidth][tWidth];
 4    __shared__ float Nds[tWidth][tWidth];
 5    int tx = threadIdx.x;
 6    int ty = threadIdx.y;
 7    int Col = blockIdx.x * tWidth + tx;
 8    int Row = blockIdx.y * tWidth + ty;
 9
10    float Pvalue = 0;
11    for (int m = 0; m < N/tWidth; ++m) {
12      Mds[ty][tx] = Md[Row*N + (m*tWidth + tx)];
13      Nds[ty][tx] = Nd[Col + (m*tWidth + ty)*N];
14      __syncthreads();
15
16      for (int k = 0; k < tWidth; ++k)
17        Pvalue += Mds[ty][k] * Nds[k][tx];
18      __syncthreads();
19    }
20    Pd[Row * N + Col] = Pvalue;
21  }
```

Figure 4. Kernel in CUDA of matrix multiplication using shared memory and coalesced accesses

Like shared memory is limited in GPU architectures, the implementations of matrix multiplication with shared memory must be tiled. This technique of splitting our problem domain into phases is called "tiling". The size of the tiled is proportional of the thread per blocks. The tiled process is executed `tWidth` times to guarantee that each thread can access all elements necessary to perform its part of the multiplication.

The block barrier synchronization before the inner loop guarantees that all data was loaded in the shared memory before the calculations are performed, while the barrier after it guarantees that the shared memory can be safely overwritten. Intra-blocks synchronizations have a small latency due to warp scheduler optimizations and processing time is not influenced by these synchronizations [32]. Consequently, we can safely ignore these intra-blocks synchronizations.

The devised value of λ for optimization (#3) is 19 and for optimization (#4) is 67. We used the same λ values for the five tested GPUs with Kepler architecture (compute capability 3.0 and 3.5). For the GT-630 GPU, which has a hybrid architecture between Fermi and Kepler, the λ values for optimizations (#1), (#2), (#3), (#4) were 3.5, 37, 22 and 96 respectively. Note that higher optimizations levels results in larger λ values and, consequently, the λ can bwe considered an estimator of the application optimization level.

B. Maximum Subarray Problem

Let X be a sequence of N integer numbers $(x_1, x_2, ..., x_N)$. The maximum subarray problem consists of finding the contiguous subarray within X which has the largest sum of elements. The solution for this problem is frequently used in computational biology for gene identification, analysis of sequence of protein and DNAs, identification of hydrophobic regions, among others. The maximum subarray problem can be solved with $O(N)$

```
 1  __global__ void subSeqMax(int *vet, int *vetFinal,
 2                           int ElemPorThread, int N_Block){
 3    __shared__ int *p;
 4
 5    for(j = 0; j < (N / t); j++){
 6      p = loadSharedmemory(vet,N_Block,j);
 7      __syncthreads();
 8      processIntervalSequence(vetFinal);
 9    }
10    writeResultingVectors(vetFinal);
11  }
```

Figure 5. Kernel simplified in CUDA of sequence maximum problem [6]

comparison operations [33] and a parallel solution for this problem was developed using Coarse Grained Model [34], resulting in $O(N/t)$ comparisons, where t is the number of threads.

In the used implementation, we have created a kernel with 4096 threads divided in 32 thread blocks with 128 threads on each. The N elements are divided in intervals of N/t elements, one per block and each block receive a portion of the array. The blocks use the shared memory for storing segments of its interval, which are read from the global memory using coalesced accesses. Each interval is reduced to a set of 5 integer variables, which are stored in vector of size $5 \times t$ in global memory. This vector is then transferred to the CPU main memory RAM for later processing.

Figure 5 shows a simplified version of the used algorithm. In line 5, each thread perform a loop which is iterated for every element allocated to the thread. On line 6, function `loadSharedMemory` loads data from the global memory to the shared one using groups of 32 threads and coalesced access. Function `processIntervalSequence` then processes the next elements from its vector interval, which updates `vetFinal` with the 5 output parameters of the interval. This function uses nested conditionals, which results in some divergence between threads of the warp. `writeResultingVectors` finally copies the `vetFinal` data to the CPU memory.

Functions `processIntervalSequence` and `loadSharedmemory` are called `N_Block/t` times, and function `writeResultingVectors` is called once. `N_Block` is the portion that each block received of the sequence. The number of cycles used per thread in this algorithm varies depending on the statical properties of the input. For the inputs that we generated, we found that a value of 100 cycles per thread, multiplied by the input size of each thread worked reasonably well.

For the maximum subarray problem, we have fixed λ equal to 0.56 for the five GPUs with Kepler architecture. For the GT-630, which has a different architecture, we used $\lambda = 1.1$.

VI. Experimental Results

We have performed experiments to evaluate the predictions of our model, by comparing these predictions with measurements of executions of the applications over 6 different GPUs: GeForce GT-630, GeForce GTX-660, GeForce GTX-680, GTX-Titan, Tesla-K20 and Tesla-K40. Five GPUs have Kepler architecture (compute capability 3.X) and the GT-630 board has a mixed Fermi and Kepler architecture (compute capability 2.1). Table I shows their specifications.

Table I
Hardware characteristics of GPUs were used for the experiments

Model	C.C.	GM	BW	SM/Cores	Clock
GT-630	2.1	2 GB	21.3 GB/s	2/96	1620 Mhz
GTX-660	3.0	2 GB	144.2 GB/s	5/960	1058 Mhz
GTX-680	3.0	2 GB	192.2 GB/s	8/1536	1006 Mhz
GTX-Titan	3.5	6 GB	288.3 GB/s	14/2688	876 Mhz
Tesla-k20	3.5	4 GB	208 GB/s	13/2496	706 Mhz
Tesla-k40	3.5	12 GB	276.5 GB/s	15/2880	745 Mhz

The number of computation ($Comp$) and communication (g_{SM}, g_{GM}, g_{L1} and g_{L2}) steps were extracted from the application source codes, and information about cache hits in cache L1 and L2 were extracted from profiling. We also confirmed the usage of FMA and SFU using profiling.

For all simulations, we considered 5 cycles for latency in the communication in shared memory and 500 cycles are considered for latency communication in global memory [9]. We used the same latency from shared memory to the L1 cache and half the global memory latency for L2 cache. For both computational and communication costs, we divide the total cost for all threads by the clock rate R and the number of cores in each GPU.

Finally, for the parameter λ, which captures the effects of thread divergence, global memory access optimizations, and shared memory bank conflicts, we used the values described in the previous section.

The source code for the experimental part of the work is available at https://github.com/marcosamaris/BSPGPU.

A. Matrix Multiplication

We used four optimization modes for the matrix multiplication application: (#1) global memory only; (#2) global memory with coalesced accesses; (#3) shared memory; and (#4) shared memory with coalesced accesses to global memory. We measured the execution time using 6 different GPUs and different matrix sizes $N \times N$. For each experiment we used the mean execution time of 10 executions.

We compared the measured times (T_m) with the times predicted by the proposed model (T_k), and used the ratio T_k/T_m to define the precision of the prediction. We used T_k values computed as described in Section V.

Figure 6 shows the obtained results. For most cases, the predicted execution time was within 10% of the measured time (T_k/T_m between 0.9 and 1.1). We consider this an excellent result, considering that we used the same λ and L1 and L2 cache hits for the five Kepler boards. The most important exception was for Optimization (#1) in the GT-630 board, which varied between 0.6 and 1.4 of the predicted time. Using the CUDA profiler we verified that the usage of L1 cache is high for this board with this optimization scheme, opposed to other configurations, where cache usage was low. Cache effects are hard to predict and are more important for non-optimized applications that, for example, do not use the shared memory and coalesced access to the global memory.

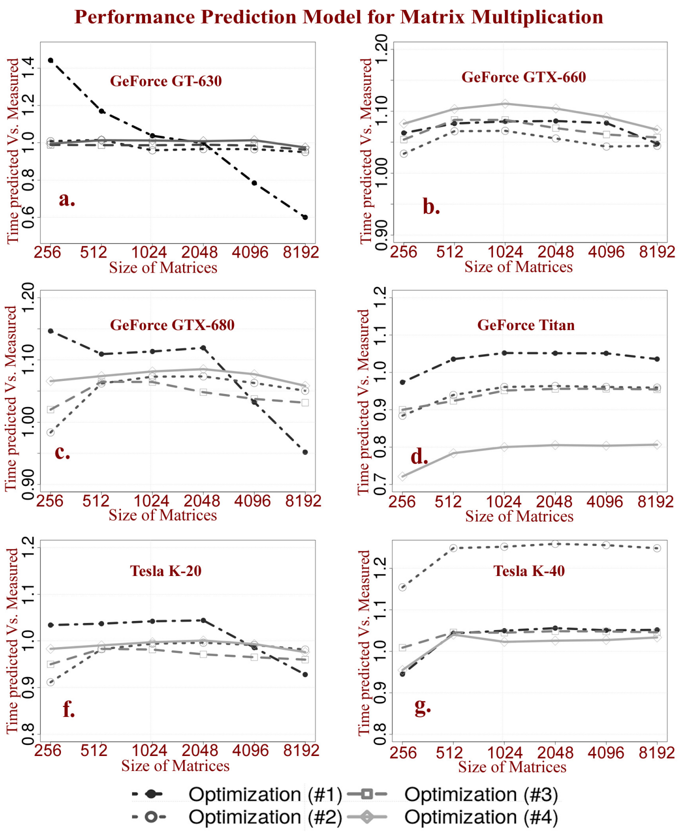

Figure 6. T_k/T_m of four optimizations of matrix multiplications over 6 different GPUs

To determine if our model can capture the required parameters to predict the execution time of application, we evaluated the case where we could adjust the λ independently for each GPU. The only exception was the model for optimization (#1) on the GT-630 board, were we adapted the parameter L1 hit rate, which was in the range of 65% and 10%. Figure 7 shows that the rate between the predicted and measured times for all the optimization of matrix multiplication in all the GPUs were between 0.9 and 1.1. The values of λ used for each simulation are show in the Table II. In all cases, the error did not increase or decrease significantly with different matrix sizes, which permits a precise performance prediction for all sizes using a single performance measurement.

These results show that we can use the model in the scenarios with different GPU types of the same architecture and with only one GPU type. In both cases the model can predict applications execution time from measurements on a single board with a single input size. With different GPU types, the prediction is less precise, since the optimal λ value is different for each board. But it can still produce adequate predictions. When using a single board, the prediction errors were always below 5%.

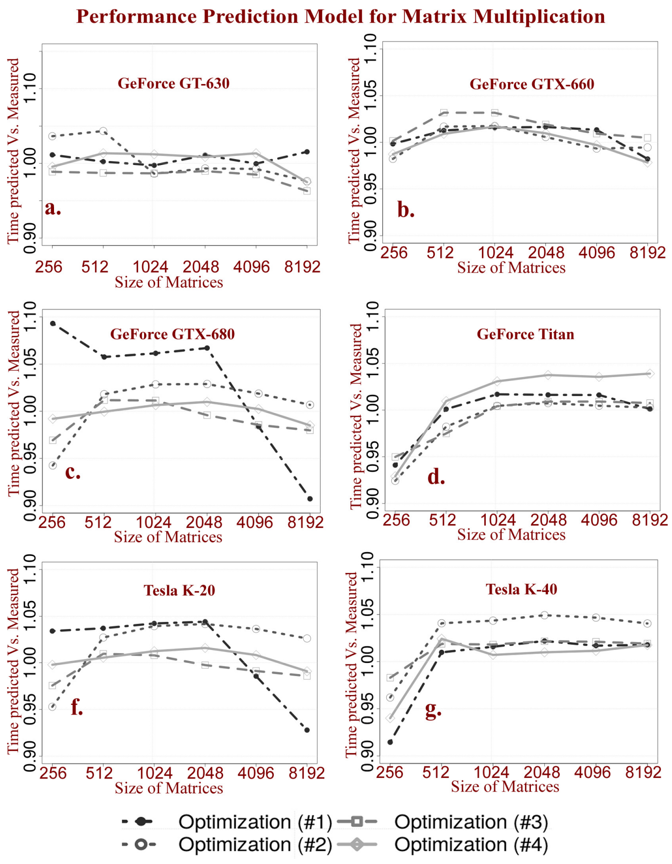

Figure 7. T_k/T_m of four optimizations of matrix multiplications with different values of λ, see table II

B. *Maximum Subarray Problem*

We compared the execution time predictions of our model with the measured execution times for the maximum subarray problem. Similarly to the matrix multiplication application, we used the same value $\lambda = 0.56$ for all boards with Kepler architecture, and used a different value $\lambda = 1.1$ for the GT-630 board.

Figure 8a shows that the rate between the predicted and measured times for the all the GPUs were between 0.8 and 1.2, showing a good prediction capability of the model. Moreover, this ratio remained nearly constant for all input sizes, which shows that the prediction accuracy is independent of the problem size. With the GT-630 board the difference was larger, between 0.8 and 1.2. We should note

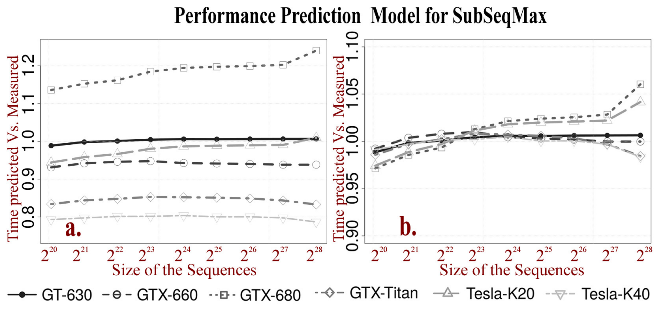

Figure 8. **a.** T_k/T_m of SubSeqMax in all GPUs. **b.** T_k/T_m of SubSeqMax in all GPUs with different values of λ, see table II

that we use a large range of input sizes, from 2^{20} to 2^{28}, resulting in a difference of 256 times between the smallest and largest inputs.

We also evaluated the scenario where we used different λ values for different boards, with the used values shown in Table II. Figure 8b shows the results using these different λ. Similarly to the matrix multiplication, the predicted execution times were closer to the measured times, with predictions errors inferior to 5% in nearly all cases and this error were stable with different matrix sizes. This scenario is less general than using the same λ values for all GPUs, but is applicable in the case were the user needs to predict the execution times using different input sizes on a single board. The presented model is simple enough in order to be clearly understood and applied to different types of GPU-based parallel executions.

Table II
VARIATION OF THE PARAMETER λ FOR EACH ONE OF APPLICATIONS IN THE GPUS USED

	Opt. (#1)	Opt. (#2)	Opt. (#3)	Opt. (#4)	SubSeqMax
GT-630	3.5	37	22	96	1.1
GTX-660	4.8	21	20	70	0.61
GTX-680	4.15	24	20	72	0.76
GT-Titan	4.5	22	18	52	0.55
Tesla-K20	4.35	22	18.5	66	0.63
Tesla-K40	4.65	24	19.5	65	0.52

VII. Conclusions and Future Works

In this work, we proposed a BSP-based model for predicting the performance of GPU applications. The BSP model offers a solution to tackle parallel problems in massively parallel architectures. We used the model to predict the performance of matrix multiplication with four different optimization levels and a coarse grained solution of subsequence maximum problem. All the applications were developed in CUDA and they were executed on 6 different GPU boards.

By considering two levels of memory, shared and global memories, we could accurately model the performance of these applications using several GPU models and problem sizes. The usage of two adaptable parameters λ was sufficient to model the effect of data coalescing during read and write operations to the global memory. A similar set of parameters also model the effects of cache hits, computation and communication process of any GPU application. In the majority of the scenarios, the time measured were around 0.8 to 1.2 times the model predicted execution time.

As future work, we will consider the scenario of multiple kernels and multiple GPUs, where global synchronization among kernels and one extra memory level, the CPU RAM, need to be considered. We presented a simple analytical model to predict performance of GPUs applications with optimal accuracy. Another future work will focus on improving statistical learning algorithms for performance prediction of applications accelerated with GPUs.

Acknowledgment

We would like to thank Cleber Silva Ferreira da Luz for the source-code of the maximum subarray problem. Thanks to the anonymous reviewers for their helpful comments and to NVIDIA Corporation who donate us a GPU Tesla K-40. This project was grant-aided by São Paulo Research Foundation (FAPESP) (procs. #2012/03778-0, #2012/23300-7 and #2013/26644-1) by CAPES and by CNPq.

References

[1] NVIDIA, *CUDA C: Programming Guide, Version 7.*, March 2015.

[2] P. Gibbons, Y. Matias, and V. Ramachandran, "The queue-read queue-write asynchronous PRAM model," *Theoretical Computer Science*, vol. 196, no. 1–2, pp. 3–29, 1998.

[3] B. H. H. Juurlink and H. A. G. Wijshoff, "A quantitative comparison of parallel computation models," *ACM Transactions on Computer Systems*, vol. 16, pp. 271–318, Aug. 1998.

[4] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Computing Surveys*, vol. 30, pp. 123–169, June 1998.

[5] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, pp. 103–111, Aug. 1990.

[6] C. Silva, S. Song, and R. Camargo, "A parallel maximum subarray algorithm on gpus," in *5th Workshop on Applications for Multi-Core Architectures (WAMCA 2014). IEEE Int. Symp. on Computer Architecture and High Performance Computing Workshops*, (Paris), pp. 12–17, 2014.

[7] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[8] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, Mar. 2008.

[9] NVIDIA Corporation, *CUDA C Best Practices Guide*, August 2014.

[10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235–246, March 2010.

[11] N. Corporation, " [Web site: Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110] Visited on Nov, 2014."

[12] P. H. Ha, P. Tsigas, and O. J. Anshus, "The Synchronization Power of Coalesced Memory Accesses.," in *DISC* (G. Taubenfeld, ed.), vol. 5218 of *Lecture Notes in Computer Science*, pp. 320–334, Springer, 2008.

[13] J. S. Kirtzic, *A Parallel Algorithm Design Model for the GPU Architecture*. PhD thesis, Richardson, TX, USA, 2012. AAI3547670.

[14] A. Goldchleger, A. Goldman, U. Hayashida, and F. Kon, "The implementation of the bsp parallel computing model on the integrade grid middleware," in *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*, MGC '05, (New York, NY, USA), pp. 1–6, ACM, 2005.

[15] R. Y. de Camargo, A. Goldchleger, F. Kon, and A. Goldman, "Checkpointing BSP parallel applications on the InteGrade Grid middleware," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 6, pp. 567–579, 2006.

[16] F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable parallel geometric algorithms for coarse grained multicomputers," in *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, (New York, NY, USA), pp. 298–307, ACM, 1993.

[17] J. Holl, B. McColl, M. Goudreau, *et al.*, "Standard: BSPlib: the BSP Programming Library," *Parallel Computing*, vol. 24, pp. 1947–1980, 1998.

[18] Q. Hou, K. Zhou, and B. Guo, "BSGP: bulk synchronous GPU programming," *ACM Transaction on Graphics*, p. 12, 2008.

[19] L. G. Valiant, "A bridging model for multi-core computing," *Journal of Computer and System Sciences*, vol. 77, pp. 154–166, Jan. 2011.

[20] K. Kothapalli, R. Mukherjee, M. Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan, "A performance prediction model for the CUDA GPGPU platform," in *High Performance Computing (HiPC), 2009 International Conference on*, pp. 463–472, 2009.

[21] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, (New York, NY, USA), pp. 114–118, ACM, 1978.

[22] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 152–163, June 2009.

[23] Y. Zhang and J. Owens, "A quantitative performance analysis model for GPU architectures," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 382–393, Feb 2011.

[24] A. Kerr, E. Anger, G. Hendry, and S. Yalamanchili, "Eiger: A framework for the automated synthesis of statistical performance models," in *High Performance Computing (HiPC), 2012 19th International Conference on*, pp. 1–6, Dec 2012.

[25] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 84:1–84:11, IEEE Computer Society Press, 2012.

[26] S. Liu, C. Eisenbeis, and J.-L. Gaudiot, "Speculative execution on gpu: An exploratory study," in *Parallel Processing (ICPP), 2010 39th International Conference on*, pp. 453–461, Sept 2010.

[27] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.

[28] D. Sampaio, R. M. d. Souza, S. Collange, and F. M. Q. a. Pereira, "Divergence analysis," *ACM Transactions on Programming Languages and Systems*, vol. 35, pp. 13:1–13:36, Jan. 2014.

[29] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," *SIGPLAN Not.*, vol. 45, pp. 105–114, Jan. 2010.

[30] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the memory hierarchy of modern gpus," in *Network and Parallel Computing* (C.-H. Hsu, X. Shi, and V. Salapura, eds.), vol. 8707 of *Lecture Notes in Computer Science*, pp. 144–156, Springer Berlin Heidelberg, 2014.

[31] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-coalesced Memory Accesses on GPU," *SIGPLAN Not.*, vol. 48, pp. 57–68, Feb. 2013.

[32] W.-c. Feng and S. Xiao, "To GPU synchronize or not GPU synchronize?," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pp. 3801–3804, May 2010.

[33] J. L. Bates and R. L. Constable, "Proofs As Programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 113–136, Jan. 1985.

[34] C. E. R. Alves, E. Cáceres, and S. W. Song, "BSP/CGM Algorithms for Maximum Subsequence and Maximum Subarray.," in *PVM/MPI* (D. Kranzlmüller, P. Kacsuk, and J. Dongarra, eds.), vol. 3241 of *Lecture Notes in Computer Science*, pp. 139–146, Springer, 2004.