A Performance Prediction Model for the CUDA GPGPU Platform

Kishore Kothapalli Rishabh Mukherjee M. Suhail Rehman
Suryakant Patidar P. J. Narayanan Kannan Srinathan
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India – 500 032.
Email:{kkishore@, rishabh_m@research., rehman@research.} iiit.ac.in
{skp@research., pjn@, srinathan@} iiit.ac.in

Abstract

The significant growth in computational power of modern Graphics Processing Units (GPUs) coupled with the advent of general purpose programming environments like NVIDIA's CUDA, has seen GPUs emerging as a very popular parallel computing platform. Till recently, there has not been a performance model for GPGPUs. The absence of such a model makes it difficult to definitively assess the suitability of the GPU for solving a particular problem and is a significant impediment to the mainstream adoption of GPUs as a massively parallel (super)computing platform.

In this paper we present a performance prediction model for the CUDA GPGPU platform. This model encompasses the various facets of the GPU architecture like scheduling, memory hierarchy, and pipelining among others. We also perform experiments that demonstrate the effects of various memory access strategies. The proposed model can be used to analyze pseudo code for a CUDA kernel to obtain a performance estimate, in a way that is similar to performing asymptotic analysis. We illustrate the usage of our model and its accuracy with three case studies: matrix multiplication, list ranking, and histogram generation.

1. Introduction

Over the past decade, the processing power of the Graphics Processing Units (GPUs) has increased tremendously. The latest GPU from Nvidia, GeForce GTX280, has a raw computing power of close to one TFLOP at a cost of about $400. Given this enormous computational power, researchers have started looking at ways to utilize this efficiently for non-graphics based applications also. This is termed as GPGPU (General Purpose Graphics Processing Units).

To this end, Nvidia now supports a C like programming language called CUDA [19] (Compute Unified Device Architecture) that allows a programmer to explicitly request that certain portions of the code be run on the GPU [1].

1. Another popular vendor of GPUs, AMD similarly supports a combination of a low-level interface, the Compute Abstraction Layer (CAL) and extensions to Brook.

The advent of CUDA has led to several high-speed implementations on the GPU. A few prominent ones are mentioned below. Image processing and filtering algorithms are studied in [13], [27], [17]. Graph algorithms such as BFS, shortest paths, graph cuts, etc. for large graphs are reported on the GPU [26], [10]. Data parallel primitives such as parallel-prefix scan[24], [18], reduction, and sorting [24] have also been studied. GPUs have also been used to implement numerical algorithms such as the FFT [8].

Speed-up of the order of 200 over conventional CPU implementations are reported for certain problems such as the n-body simulation[17, Chapter 31]. At the other extreme, there are problems where the best reported speedup is just about 3 [15]. While the speed-up reported does depend on the hardware available at the time it is reported, it is in general not possible to explain the origins of the speed-ups or the varying performance across different problems. Also the relationship between different facets of these implementations is not clearly understood. As such it is hard to adopt a structured approach to optimize these implementations. These problems are compounded by the fact that leading manufacturers of GPUs do not divulge detailed low-level architectural details of their product.

Thus, there is a significant need to understand the computational abilities of modern-day GPUs so as to use them efficiently. When one considers multiprocessor architectures, there are several issues that one has to contend with apart from computations, viz. the memory hierarchy, the interprocess communication, synchronization, and the like. When working at a purely algorithmic level, and ignoring the effects of memory hierarchy, cost of synchronization, etc., the PRAM model [5] has been a highly successful model and can give tight asymptotic bounds on the runtime and the total work done. But, the abstraction of the PRAM model does not help one to choose a right algorithm for a given architecture, for example the GPU. Hence, it is of interest to see how much of architectural details should be modeled carefully so that one can work at a level of abstraction that can be used to analyze algorithms and at the same time make reasonably accurate claims.

The benefits of such a model are manifold, some of which are given below. Firstly, it helps augment the

978-1-4244-4921-7/09/$25.00 ©2009 IEEE

PRAM model to understand the limits of parallelizability of algorithms on the GPUs. A second benefit is to provide an informative profile of a GPU program so as to be able to identify bottlenecks. As in the case of sequential architectures, we feel that a good simulator is the need of the hour when one wishes to evaluate the effect of certain design choices for future versions of GPUs. Our model can help efforts in this direction.

In this paper, we propose a model for the same. Our model coupled with relevant case-studies shall formalize several aspects of GPU programming that serves to bridge the gap between the algorithmic developments and the application engineering. The focus of our work is to explain the behavior of the GPU and additionally understand the nature of problems that scale well on the GPU. Using our model, one can make educated claims about a program in execution on the GPU.

1.1. Related Work

The parallel algorithms community has developed several models to design and analyze parallel algorithms such as the network model [14], the PRAM model [5], the Log-P model [4], the QRQW model [6], [7], among others. These models help the algorithm designer to exploit the parallelism in the problem. However, as these are mostly architecture independent they fail to help the algorithm designer leverage any specific advantages offered by the architecture or work around the ill-effects of any architectural limitations on the performance of the algorithm.

As far as modeling for performance on multiprocessor architectures is concerned, there is very little reported work. In [22], the authors discuss the parameter space and present ways to prune the size of such a space and get a highly optimized code. The result is a gain of about 17% for a particular medical imaging application [22]. However, our work does not target code or runtime optimizations. An extension of this work appears in [23] where the authors consider the multi-GPU design space optimization. However, they need a model to predict the baseline implementation on a single GPU. Our work can exactly fill that need. So we place our work as complementary to that of [23].

A model similar to ours appears in [16] where also the authors rely on separating memory and compute requirements. But their model is applicable only for a class of programs called ”Iterative Stencil Loops”. Very recently, Hong and Kim independently presented a work [12] that also predicts the runtime of a kernel on the GPU. They consider a set of 23 parameters that can be used to predict the runtime of a kernel on the GPU. Despite a similar approach, we find a few differences. Firstly, they do not try to model the GPU in terms of already exisitng parallel models. We feel that this is important so that the already avaialble knowledge base can be reused. Secondly, as we relate the GPU to the BSP model, we are naturally able to model an entire program instead of a single kernel.

1.2. Our Results

In this paper, we propose a fairly complete performance model for the Nvidia GPU. Our model tries to abstract the GPU computational model by considering important features of the present generation Nvidia GPU GTX 280. We use a combination of the BSP model of Valiant [25], the PRAM model of Fortune and Wyllie [5], and the extension to PRAM model proposed by Gibbons et al. called the QRQW model [7]. We note that none of the models individually can explain the behavior of a GPU. We also note that slight modifications are required to these models to model GPU computations accurately.

The proposed model outlines the relationship between the various components of the GPU architecture like the number of cores, effects of memory latency, memory access conflicts, cost of computing, scheduling, pipelining, etc. This model can be used to analyze pseudo-code for a CUDA kernel and finally predict the performance, almost analogous to the way asymptotic analysis is carried out in the case of sequential computing.

We devise experiments that showcase the effects of memory access related issues like coalescing and bank conflicts and the corresponding latency penalties incurred. We further demonstrate the use of our model on three real-world parallel algorithms - matrix multiplication, list ranking, and histogram generation. These case studies have been chosen so that one of them is compute intensive, one is (global) memory intensive, and one is shared memory based. Thus, these three case studies cover the entire scope of the proposed model.

1.3. Organization of the Paper

The rest of the paper is organized as follows. In Section 2, we provide a basic introduction to the GPU computational model. In Section 3 we describe the proposed performance model. Section 4 corroborates the proposed model using targeted experiments. This is followed by three case studies in Section 5. The paper ends with some concluding remarks after mentioning a few limitations of our model.

2. GPU Architecture and CUDA

Nvidia’s unified architecture (see also Figure 1) for its current line of GPUs supports both graphics and general computing. In general purpose computing, the GPU is viewed as a massively multi-threaded architecture containing hundreds of processing elements (*cores*). Each core comes with a four stage pipeline. Eight cores, also known as *Symmetric Processors* (SPs) are grouped in an SIMD fashion into a *Symmetric Multiprocessor* (SM), so

that each core in an SM executes the same instruction. The GTX280 has 30 such SMs, which makes for a total of 240 processing cores. Each core can store a number of thread contexts. Data fetch latencies are tolerated by switching between threads. Nvidia features a zero-overhead scheduling system by quick switching of thread contexts in the hardware.

The CUDA API allows a user to create large number of threads to execute code on the GPU. Threads are also grouped into *blocks* and blocks make up a *grid*. Blocks are serially assigned for execution on each SM. The blocks themselves are divided into SIMD groups called *warps*, each containing 32 threads on current hardware. An SM executes one warp at a time. CUDA has a zero overhead scheduling which enables warps that are stalled on a memory fetch to be swapped for another warp.

The GPU also has various memory types at each level. A set of 32-bit registers is evenly divided among the threads in each SM. 16 KB of *shared memory* per SM acts as a user-managed cache and is available for all the threads in a Block. The GTX 280 is equipped with 1 GB of off-chip *global memory* which can be accessed by all the threads in the grid, but may incur hundreds of cycles of latency for each fetch/store. Global memory can also be accessed through two read-only caches known as the *constant memory* and *texture memory* for efficient access for each thread of a warp.

Computations that are to be performed on the GPU are specified in the code as explicit *kernels*. Prior to launching a kernel, all the data required for the computation must be transferred from the *host* (CPU) memory to the GPU *global* memory. A kernel invocation will hand over the control to the GPU, and the specified GPU code will be executed on this data. Barrier synchronization for all the threads in a block can be defined by the user in the kernel code. Apart from this, all the threads launched in a grid are independent and their execution or ordering cannot be controlled by the user. Global synchronization of all threads can only be performed across separate kernel launches. For more details, we refer the interested reader to [19], [18].

3. Our Performance Model for the GPU

The model we present for the GPU is a combination of known models of parallel computation. Given the complex architecture of the GPU, it turns out that none of these models suffice individually and a combination of them along with a few extensions is required. The models we use are:

- The BSP model of Valiant [25],
- The PRAM model of Fortune and Wylie [5], and
- The QRQW model of Gibbons, Matias, and Ramachandran [6], [7].

In the following we describe our modeling of the GPU using the above three models.

3.1. Synchronization Model

As discussed in Section 2, CUDA programs are written in units called *kernels*. Threads start synchronously at the beginning of each kernel and are synchronized at the end of each kernel. Thus, the basic unit of synchronization in a CUDA program is the kernel. This fits the BSP model of parallel computing quite closely, with an implicit call to synchronize at the end of each kernel. Notice however that while in the BSP model, synchronization is at *regular* intervals of L time units, our model does away with this requirement. Given the lack of any routing infrastructure in the GPU, we rely on the BSP model only as far as the notion of super-steps [25] is concerned.

A further facet of the GPU is that threads in a block can all be synchronized explicitly within a kernel by a call to the primitive `__syncthreads()`. This puts a barrier for threads in a block and it is guaranteed that executing this call and thereby synchronizing threads in a block takes 4 cycles, plus additional wait time depending on the circumstances. But this being an explicit and optional call, threads need not be synchronized every 4 cycles.

It thus implies that the time taken by a GPU program can be expressed as the sum of the times taken by the super-steps, or kernels. For the sake of simplicity, we ignore the effect of intra-kernel synchronization steps such as `__syncthreads()` on the overall runtime.

3.2. GPU *a. la.* (QRQW) PRAM

The other parts of the GPU model are not as straightforward. We will propose a model for the GPU that accounts for its memory hierarchy along with computation.

The PRAM model is an extension of the traditional RAM model for sequential computation. (See Fortune and Wyllie [5] for an elaborate description). It does not distinguish between memory access operations and computational operations and assumes that both cost a unit of time. It also ignores other costs such as synchronization. However, present parallel computer architectures, including the GPU, have a deep memory hierarchy and/or significantly complex memory access model. Hence, it is required to address the cost of memory accesses and computational operations separately.

3.2.1. Cost of Computation. Notice that the fundamental element of computation in a GPU program is a thread in a kernel. A thread can be viewed as performing some memory reads, computations, and memory writes. To look at the cost of computation is by far the easiest. For a crude estimate one can simply treat all operations uniformly and for a unit time, or same number of cycles [16]. However, the GPU is not a very versatile architecture. The time taken by computational operations can

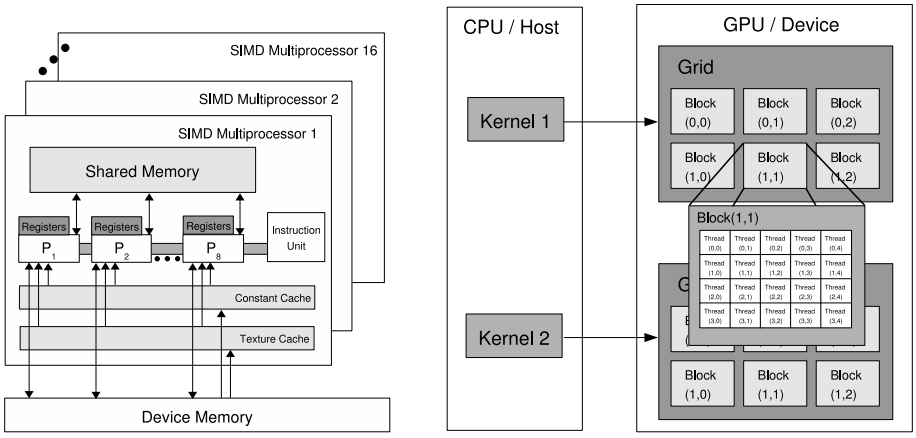


Figure 1. The CUDA Computation Model.

vary from 4 cycles for a simple addition to 16 cycles for a 32-bit integer multiplication and many more for an integer modulus. Thus, to get better results, one has to consider the cycle requirement of the computational operations in a thread.

Hence, in our model, we propose to arrive at the cycles required by the computation in a thread. For this, we can use published architectural details to see the cycles required by each operation and add them up. For example, if a thread has two integer additions and two multiplications, then it requires $2 \cdot 4 + 2 \cdot 16 = 72$ cycles [19]. The number of cycles can also be obtained as a function of the input size as is done in typical asymptotic analysis. Obtained in this fashion, let N_{comp} be the cycles required for computation in a thread.

3.2.2. Cost of Memory Accesses. There is a deep memory hierarchy in the GPU with a large variation in the access time for each level of the memory hierarchy. See Figure 1 for the available memory hierarchy. Hence, to estimate the time taken by a thread to read global memory, one has to be more careful. Two important members of this hierarchy are the *global memory* and the *shared memory*. We first consider memory accesses to global memory and then focus on the shared memory.

Accessing Global Memory. Reading/writing from/to a cell from the global memory has a cost of 400–600 cycles [19]. In our work, we take the average value of 500 cycles per read.

The above does not account for any cache or cache-like effects. The effect of spatial and temporal locality on caches in sequential computation is well understood. However, there the situation is simple as one is interested in the locality exhibited by a single program in its memory accesses. With parallel architectures such as the GPU, it is however dependent also on the locality exhibited by a set of concurrently executing threads.

Recall that on the GPU, threads are executed as a batch called a *warp*. GPU accesses global memory in contiguous chunks of 128 Bytes called a *segment*. Threads in a half-warp, i.e., half the number of threads in a warp, that are concurrently under execution benefit from inter-thread spatial locality if they access locations within a segment. In this case, one transaction of reading a segment from the global memory suffices to serve all the threads in the half-warp that exhibit inter-thread spatial locality. If a transaction benefits t threads in warp, then the average access cost per access for these t threads in this situation can be taken to be $\frac{500+t}{t}$.

This phenomenon of benefiting from inter-thread locality is called in the GPU parlance as *coalesced reads*. The effect of coalescing on data accesses is significant enough, up to a factor or 16 when 32 threads benefit from the coalescing effect. Hence, many works reported in the literature devote enough attention to optimize the program to benefit from coalescing effects (see e.g., [24], [9]).

When a thread in a half-warp accessing cells in the global memory does not benefit from inter-thread spatial locality, the access time is as high as 500 cycles per access. Here, each access translates to a separate transaction to the global memory. This is called a *non-coalesced* read in the GPU parlance and can have a significant impact of the performance of a program executing on the GPU.

Accessing Shared Memory. GPU provides a shared memory for threads which is ideally useful for frequently accessed variables that are needed by threads. This is a low-access cost memory in the hierarchy, about 4 cycles per access, but comes with several restrictions. Shared memory is of very small size (16 KB per SM) and has to be shared over *all* threads scheduled on an SM. Furthermore, if more than one thread is accessing the same bank in the shared memory at the same time, this results in a memory contention, which can increase the access cost.

In the case of a memory contention, the GPU behavior is close to that of a QRQW Asynchronous PRAM model [6], [7] with a linear cost function. If there are t threads in a warp in contention, the access cost is $4t$

cycles. However, the QRQW model as proposed in [7] is a purely shared memory based model like the PRAM. So the QRQW model alone cannot explain the GPU model in its entirety as it ignores other factors such as synchronization.

We add that, if the accesses made by threads are not deterministic, but are randomized, then one can consider the expected number of conflicts and conflicts with high probability to estimate the cost of accesses to the shared memory.

Finally, let N_{memory} be the number of cycles required for all the memory accesses by a thread. This number includes the cost of both global memory and shared memory accesses by a thread in a kernel.

3.3. Effect of Scheduling

The above model of separating memory accesses and computations works as far as a single thread is concerned. However, parallel architectures employ scheduling to hide the memory latency. It can also be inferred that the actual scheduling employed will be preemptive in nature. More details about the effect and nature of scheduling can be obtained only by knowing the actual scheduling performed inside the GPU. This, unfortunately, is not public knowledge.

Hence, we take the following approach. Let $C(T)$ denote the number of cycles required by a thread. The best effect of scheduling is to completely hide latencies. So the number of cycles required by a thread is $C(T) = \max\{N_{\text{comp}}, N_{\text{memory}}\}$. We call this the MAX model. If scheduling does not help at all in latency hiding, then the number of cycles required by a thread is $C(T) = N_{\text{comp}} + N_{\text{memory}}$. We call this the SUM model. In either case, the presence of a 4-stage pipeline in each core of the GPU has its own effect which is analyzed in the following.

3.4. The Overall Model

We now combine the ideas from the above sections to estimate the time taken by a program P in execution on the GPU. The BSP model allows us to look at time as the sum of the times across various kernels. Thus, given a CUDA program P with r kernels $K_1, K_2, \cdots, K_r$, the time taken is $\sum_{i=1}^{r} T(K_i)$ sec. where $T(K_i)$ gives the time taken by kernel K_i. Thus, we have:

$$T(P) = \sum_{i=1}^{r} T(K_i) \text{ sec.} \tag{1}$$

For a kernel K, we now have to consider the GPU execution model. Recall that *blocks* are assigned to SMs and let each block consist of N_w *warps*. Each warp consists of N_t threads and threads in each warp are executed in parallel. Though it is possible that each SM gets blocks in a batch of up to 8 blocks so as to hide idle times, this is equivalent to having all blocks execute in a serial order for the purposes of estimating the runtime. One has to also consider the fact that each of the N_c cores (or SPs) in an SM on the GPU has a D-deep pipeline that has the effect of executing D threads in parallel.

In addition, it is also required to estimate the cycle requirement of a single thread. This can be done by estimating the compute and memory access times as discussed in Sections 3.2.1 and 3.2.2. We take the approach that the number of cycles required by a kernel is the *maximum* required by some thread in that kernel. Let the maximum number of cycles required by any thread executing the kernel K be $C_T(K)$. Thus, $C_T(K)$ can be expressed as the maximum over all $C(T)$ for T a thread executing the kernel K. Therefore,

$$C_T(K) = \max_T C(T). \tag{2}$$

Notice that if we are using the MAX (SUM) model, then the $C_T(K)$ term in the above should be obtained using the MAX (*resp.* SUM) model .

Finally, the time taken for executing kernel K is estimated as follows. Let $N_B(K)$ be the number of blocks assigned to each SM in sequence in the kernel K, $N_w(K)$ be the number of warps in each block in the kernel K, $N_t(K)$ be the number of threads in a warp in the kernel K. Then, the number of cycles required for the kernel K, denoted $C(K)$ is:

$$C(K) = N_B(K) \cdot N_w(K) \cdot N_t(K) \cdot C_T(K) \cdot \frac{1}{N_C \cdot D} \tag{3}$$

To convert the cycles required to time in seconds, we have to multiply Equation (3) by the clock rate of the GPU as in the equation below, where R is the clock rate of a GPU core.

$$T(K) = \frac{C(K)}{R} \text{ sec.} \tag{4}$$

Since it is possible to have a different structure on the number of blocks, number of warps per block etc. in each kernel, we parameterize these quantities according to the kernel.

To illustrate Equations (3, 4), Figure 2 is useful. Each of the SMs in the GPU get multiple blocks of a kernel. In the picture we consider $N_B = 8$. Each of these blocks are executed on an SM by considering each block as a set of N_w warps. Each warp is then treated as a set of N_t threads. It is these threads that are essentially executed in parallel on the N_c cores of the SM. In Figure 2, we have used $N_w = 16$, $N_c = 8$, and $N_t = 32$.

Unlike sequential computation, there is another element that has an impact on the performance of GPU programs. Multiprocessors employ time-sharing as a latency hiding technique. Within the context of the GPU, this time-sharing is in the form of each SM

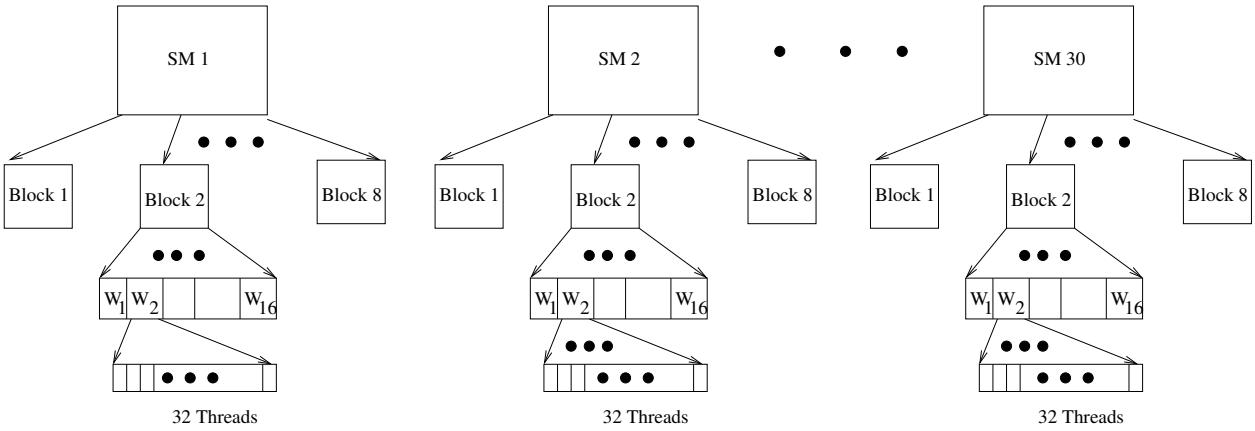


Figure 2. The threads in a warp are executed in parallel. Groups of warps are arranged into block and blocks are assigned to SMs.

handling more than one block of threads at the same time. To model this situation and its effect, let us assume that each SM gets b blocks that it can time-share. Notice that when we use the MAX or the sum model to estimate the time taken by a kernel, all the b blocks then require b times the time taken by a single block. The number of blocks assigned sequentially to an SM N_B effectively reduces by a factor of b. So there is no net effect of time sharing as long as latencies are hidden well. So, our Equation (4) stands good even in the case of time sharing.

Parameter	Definition
N_w	Number of warps per block
N_t	Number of threads per warp = 32
D	Pipeline depth of a core
N_c	Number of cores per SM
K_i	Kernel i on the GPU
$C_t(K)$	Max. number of cycles required by any thread in kernel K
R	Clock rate of GPU
$T(K)$	Time taken by a kernel K

Table 1. List of Parameters in our Model

3.5. A Few Reflections on the Model

At this stage we find it pertinent to discuss two issues related to the model. The first question the reader is likely to have is: "Is it required to model at such a low level where one has to count the number of cycles for each operation?".

The performance of a CUDA kernel can vary drastically with small changes in memory access strategies. Using shared memory may yield up to 20 times better performance than using global memory and using coalesced global memory accesses may result in as much as 5 times performance increase over non-coalesced access. Arithmetic operations also have highly varying cycle requirement such as 4 cycles for operations like integer addition to 48 cycles for integer modulus [19]. Any model that does not capture these changes is unlikely to be accurate.

The second question that a reader would have is: "How difficult is it to perform such an analysis?". In our view, performing such an analysis for arithmetic operations would be not be significantly harder than performing an asymptotic analysis. Unlike other architectures, the GPU does not have any sort of implicit caching across different types of memories. Storing data in a particular type of memory, and then, the strategy to access it is the explicit choice of the user. As there is no scope for issues like cache misses, analyzing memory access patterns in our model is no easier or harder than it is in the case of asymptotic analysis.

4. Corroborating the Model

We show the results of some basic experiments that corroborate our model. The first experiment deals with effects of memory coalescing and the second deals with memory bank conflicts.

4.1. Experiment 1: Coalesced/Non-coalesced Access

We set up an experiment that controls how many threads in a warp can benefit from a coalesced access. This is controlled by the parameter `stride` in the code Listing 1. (see Appendix A). `stride` denotes the gap between the elements that are accessed in sequence by a single thread. Hence, threads in a half-warp can benefit from a coalesced access if the value of `stride` is large. For example, when `stride` = 32, each thread of a warp gets consecutive elements, which ensures complete coalescing. When the `stride` is 1, each thread across a warp gets elements that are displaced by 32, hence is guaranteed to be completely non-coalesced and requires 16 memory transactions to be serviced for a half-warp. In order to ensure a fair comparison, in our

code, the number of accesses by a thread is independent of `stride`.

In the code given in Listing 1, the amount of computation per iteration is very small compared to the memory access latency for `stride = 1`. However, as we increase the value of `stride`, memory access and computation take about the same number of cycles. Using the MAX model, we predict the runtime of this kernel and plot it along with the actual runtime in Figure 3(a) plots the program runtime for various values of `stride`. It must be noted that a purely memory access base code, i.e., with little compute, is difficult to model due to limited knowledge about the memory access hardware.

4.2. Experiment 2: Understanding access conflicts

In this experiment, keeping the overall structure of the global accesses as in Experiment 1, each thread now writes an element to the shared memory. The access pattern to the shared memory is controlled by a variable `bank` which can be given a value between 0 to 16. With a larger value of `bank` we can thus increase the number of bank conflicts.

The kernel in Listing 2 (in Appendix A) has about 16 cycles of compute per iteration and there are 64000 iterations. The number of cycles required for memory is about `bank`$\times 4$ per iteration. The actual runtime and the runtime predicted by our model is plotted in Figure 3(b). As can be seen, there is indeed a linear dependency on the number of conflicts and the program runtime.

5. Case Studies

In this section, we further validate our model by considering non-trivial problems as case studies. The case studies we consider are matrix multiplication, list ranking, and histogram generation. These case studies cover all the features of our model. The matrix multiplication kernel is compute intensive, the list ranking kernel is global memory intensive and is a popular case study for irregular algorithms. The histogram kernel makes use of shared memory resulting in bank conflicts. Hence, the choice of our case studies is justified. As only a single kernel is anaylzed in each of our case studies, for simplicity, we drop the parameter K in quantities such as $N_B(K)$ and simply write N_B.

5.1. Case Study 1: Matrix Multiplication

Matrix multiplication is a highly popular problem in parallel computing with several applications. The algorithm considered here [19, Chapter 6] launches one thread per element of the product matrix C in $A \times B = C$. To improve data locality, we can keep a block of rows from the matrix A and a block of columns from the matrix B in the shared memory. These blocks can be multiplied to get partial results. As the access to elements in the shared memory can be made to be non-conflicting, by choosing the access pattern carefully [19], the algorithm benefits from fast accesses to the shared memory as well as maintaining coalesced access from global memory.

For multiplying matrices of size $N \times N$, the total number of blocks is $\frac{N^2}{256}$ with each block consisting of $N_w = 8$ warps with each warp consisting of $N_t = 32$ threads. So each SM gets $N_B = N^2/(256 \cdot 30)$ blocks.

As per the implementation above [19], the work done per block/thread scales with the number of rows and columns. The dimensions of a thread block are 16×16. Each thread then loads a value from matrices A and B into shared memory, iteratively computes each element of C_{sub} and writes it back to memory. This requires $\frac{N}{16}$ iterations for each thread. The number of computation cycles required per thread for a matrix of N rows and N columns can be counted to be $N_{\text{comp}} = 760N/16$.As each thread performs three global and two shared memory accesses per element computed, the cycles spent in memory operations in this thread can be counted to be $N_{\text{memory}} = 240N/16$.

Thus, when using the MAX model, the compute time dominates the memory time. Let $C_T = \max\{N_{\text{comp}}, N_{\text{memory}}\}$. Using Equation 4, the total time required for multiplying the matrices under the MAX model is:

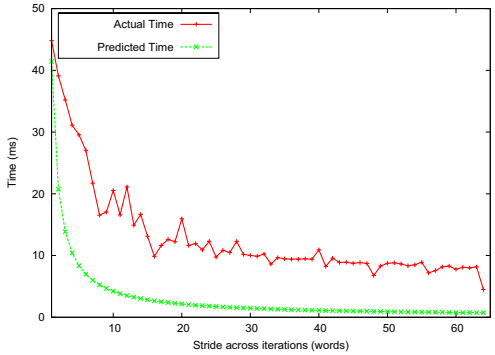
$$\left\lceil \frac{N^2}{256 \cdot 30} \right\rceil \cdot 8 \cdot 32 \cdot \frac{760N}{16} \cdot \frac{1}{32 \times 1.3 \times 10^9} \text{ sec.}$$

At $N = 128$, the estimated time using the MAX model comes to around 0.11 ms which compares favorably with the actual time of 0.16 ms. The predicted run time of this algorithm for both the SUM and MAX models for various values of N are plotted in Figure 4(a). Matrix multiplication requires block synchronization which is difficult to predict and hence there is a some deviation from the actual runtime.

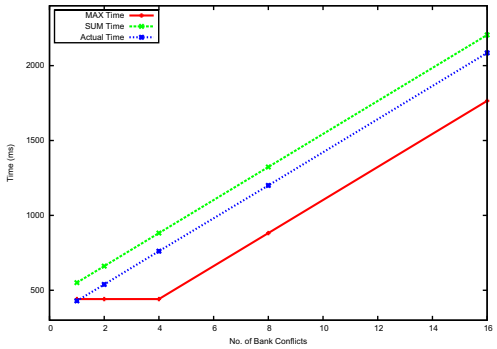
5.2. Case Study 2: List Ranking

In parallel computing, list ranking is one of the fundamental operations with applications to several problems. While list ranking does not figure at all as an important problem in sequential computing, the difficulty of the problem in parallel computing is recognized early by Wyllie [14]. Using various techniques, several algorithms to solve this problem are proposed [11], [1], [3].

For symmetric multiprocessors, Hellman and JàJà [11] proposed an algorithm for list ranking that has a runtime of $O(\log N)$ for a list of N elements with high probability, when the number of processors is small compared to the size of the input. Their algorithm suggests that the N/p sublists are ranked locally and a list of size N/p be ranked sequentially. Several implementations of this algorithm are reported on various multi-core architectures

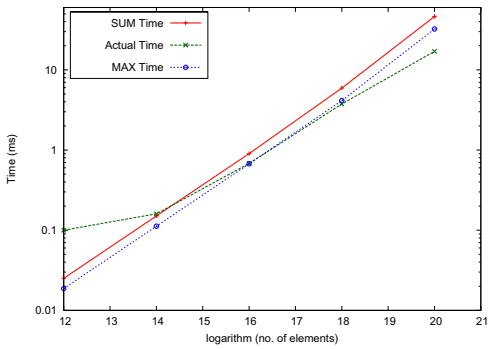


(a) Effect of Memory Coalescing

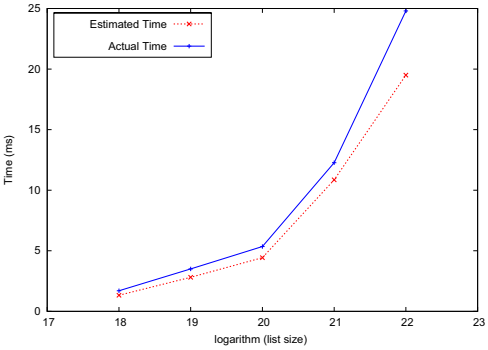


(b) Effect of Bank Conflicts

Figure 3. Studies to corroborate the model.



(a) Matrix Muliplication



(b) List Ranking

Figure 4. Figure (a) shows the estimated and actual times of the matrix multiplication kernel on square matrices of size $2^6 \times 2^6$ to $2^{11} \times 2^{11}$. Figure (b) shows the estimated and the actual time taken by the list ranking kernel on lists of size varying from 2^{18} elements to 2^{22} elements.

including the most recent one on the Cell BE [2].

A recursive variant of the algorithm, developed for GPU, proceeds as follows [21]. Initially, p splitters (local leaders) are chosen at equi-distant points in the successor array. Using these splitters, elements of the list are ranked locally so that each splitter ranks the elements till the next splitter is reached. Now, recursively, the list of splitters is ranked. Finally, the (global) ranks are computed using the local rank and the rank of the splitter to which the element belongs.

In this case study, we focus on the local ranking aspect of the recursive variant of the Hellman-JàJà [11] algorithm. During the local ranking phase, it can be noticed that the elements accessed each thread exhibit no particular spatial locality. Hence, this kernel falls under the simple model of the GPU as a PRAM with non-coalesced accesses to the global memory. For a cmplete discussion, we refer the reader to [21].

With N elements and $p = \frac{N}{\log N}$ splitters, we require $\frac{N}{\log N}$ threads. These threads are grouped into $\frac{N}{512 \log N}$ blocks of 512 threads each. Of these at most $N_B = \left\lceil \frac{N}{512 \cdot 30 \cdot \log N} \right\rceil$ blocks are assigned to any single SM on the GPU. Each of these blocks consists of $N_w = 16$ warps of $N_t = 32$ threads each.

Given a random list as an input, it is likely that some threads process more elements compared to other threds. Typically, the most likely size of the sublist was observed to be $4 \log N$ elements, which also confirms to known results on probability. The memory cycles taken by a thread can be computed as follows. Each thread involves three reads/writes to the global memory for each element that this thread is traversing. All these accesses tend to be non-coalesced. So, with about $4 \log N$ elements per sublist, $N_{\text{memory}} = 4 \log N \cdot 3 \cdot 500$. The compute in each thread is very minimal. So we ignore this completely and set $C_T(K) = N_{\text{memory}}$ in both the SUM and the MAX models.

The overall time taken by the kernel to compute the local ranks for each sublist can then be computed using Equation 4 as:

$$\left\lceil \frac{N}{512 \cdot 30 \cdot \log N} \right\rceil \cdot 16 \cdot \frac{32}{8 \times 4} \cdot 4 \log N \cdot 3 \cdot 500 \cdot \frac{1}{1.3 \times 10^9} \text{ sec.}$$

For $N = 2^{22}$ we get the time per SM to be ≈ 21.0 millisecond. This compares favorably with the actual time of 24 millisecond for $N = 2^{22}$. Figure 4(b) shows

the comparison of the estimate and the actual times over various list sizes, ranging from 256 K to 4 M elements. We note that since the computation in each thread is very minimal compared to the memory access cost, the models MAX and SUM exhibit identical behavior. So we show only the estimates from the MAX model.

5.3. Case Study 3: Histogram Generation

Counting elements of the same category is a common problem spreading across a wide variety of applications. It is one of the basic primitives in the field of statistics, image processing, and data engineering. Let N observations be chosen independently and uniformly at random between 1 through B, both inclusive. Let us assume that these N observations are to be placed into B bins.

In this case study we test our model against the shared memory access patterns. Each block loads its share of input data one by one from the global memory in a coalesced manner and updates the histogram in the shared memory. When the block has gone through all its data, the shared memory histogram is copied to the global memory in a coalesced manner. These local histograms can then be added together to obtain the global histogram. See [20] for more details. In this case study, we only compare the estimates from our model with that of the actual time for computing the local histograms. Obtaining the global histogram from these local histograms is an easy operation and hence is omitted from the timing analysis.

For correctness, threads in a block should compute the local histogram of the block using atomic operations. However, our model at this point does not account for atomic operations. Hence, for the purposes of this case study, we let all these increment operations conflict in the shared memory. We thus time the kernel ignoring the correctness of the result.

In our implementation [20], each thread builds the local histogram of $N/(1920 \times 256)$ elements into 256 bins. This involves reading each element from the global memory in a coalesced manner and updating the count in the appropriate bin in the shared memory. Thus, the amount of compute and the memory access per element is very small.

Using the model described in Section 3, we obtained the estimates for the runtime according to both the MAX and the SUM variants. The actual and the estimated times are plotted in Figure 5. The plot suggests that latency hiding works very well in this kernel as the MAX model closely predicts the runtime.

6. Limitations of Our Model

Our model however has a few limitations. Our model does not consider the effect of intra-block synchronization calls such as `__syncthreads()`. However, the model can be extended for this by treating each kernel as being composed of sub-kernels separated by calls to `__syncthreads()`. Our model at present does not handle atomic operations. These are to be handled by serializing the threads participating in the atomic operation. Also, we did not specifically mention the effect of computational divergence among threads in a warp. Presently, we considered only two members of the memory hierarchy and did not consider accesses to texture memory and shader memory.

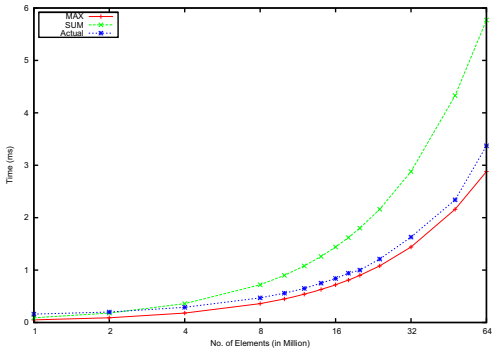


Figure 5. The estimated and the actual runtime of the histogram kernel on various input sizes.

Bringing these parameters into a future model requires a better understanding of the architecture and the scheduling aspects of the GPU.

7. Conclusions

In this paper we proposed a performance model for the Nvidia GPU by using popular models in the parallel algorithm community. Our effort is a step to bridge the gap between the theory and practice of parallel programming on the GPU. In future, we wish to use this model to develop a simulator for the GPU that can ease further architectural developments of GPGPU.

References

[1] ANDERSON, R. J., AND MILLER, G. L. A Simple Randomized Parallel Algorithm for List-Ranking. *Information Processing Letters 33*, 5 (1990), 269–273.

[2] BADER, D. A., AGARWAL, V., AND MADDURI, K. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In *Proc. of IEEE IPDPS* (2007), pp. 1–10.

[3] COLE, R., AND VISHKIN, U. Faster Optimal Parallel Prefix sums and List Ranking. *Information and Computation 81*, 3 (1989), 334–352.

[4] CULLER, D., KARP, R., PATTERSON, D., A. SAHAY, K. E. S., SANTOS, E., SUBRAMONIAN, R., AND VON EICKEN, T. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. ACM PPoPP* (1993), pp. 1–12.

[5] FORTUNE, S., AND WYLLIE, J. Parallelism in Random Access Machines. In *Proc. ACM STOC* (1978), pp. 114–118.

[6] GIBBONS, P. B., MATIAS, Y., AND RAMACHANDRAN, V. The Queue-Read Queue-Write Asynchronous PRAM model. In *In Proc. of EURO-PAR* (1996).

[7] GIBBONS, P. B., MATIAS, Y., AND RAMACHANDRAN, V. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms. *SIAM J. Comp. 28*, 2 (1999), 733–769.

[8] GOVINDARAJU, N., AND MANOCHA, D. Cache-efficient Numerical Algorithms using Graphics Hardware. *Parallel Computing 33*, 10-11 (2007), 663–684.

[9] GUTIERREZ, E., ROMERO, S., TRENAS, M. A., AND ZAPATA, E. L. Memory Locality Exploitation Strategies for FFT on the CUDA Architecture. In *Proc. of High Perf. Comp. for Comp. Sci.* (2008), pp. 430–443.

[10] HARISH, P., AND NARAYANAN, P. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing HiPC 2007* (2007), pp. 197–208.

[11] HELMAN, D. R., AND JÀJÀ, J. Designing Practical Efficient Algorithms for Symmetric Multiprocessors. In *Proc. ALENEX* (1999), pp. 37–56.

[12] HONG, S., AND KIM, H. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2009), ACM, pp. 152–163.

[13] HOPF, M., AND ERTL, T. Hardware Accelerated Wavelet Transformations. In *Proc. EG Symposium on Visualization* (2000), pp. 93–103.

[14] JÀJÀ, J. *Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[15] LUO, Y., AND DURAISWAMI, R. Canny Edge Detection on Nvidia CUDA. In *Proc. of IEEE Computer Vision and Pattern Recognition* (2008), pp. 1–8.

[16] MENG, J., AND SKADRON, K. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing* (New York, NY, USA, 2009), ACM, pp. 256–265.

[17] NGUYEN, H. *GPU Gems 3*. Addison-Wesley Professional, 2007.

[18] NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. Scalable Parallel Programming with CUDA. *ACM Queue 6*, 2 (2008), 40–53.

[19] NVIDIA CORPORATION. CUDA: Compute Unified Device Architecture Programming Guide. Tech. rep., 2007.

[20] PATIDAR, S., AND NARAYANAN, P. J. Scalable Split and Gather Primitives for the GPU. Tech. Rep. IIIT/TR/2009/99, IIIT-Hyderabad, 2009.

[21] REHMAN, M. S., KOTHAPALLI, K., AND NARAYANAN, P. J. Fast and Scalable List Ranking on the GPU. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing* (New York, NY, USA, 2009), ACM, pp. 235–243.

[22] RYOO, S., RODRIGUES, C. I., STONE, S., BAGHSORKHI, S. S., UENG, S.-Z., STRATTON, J. A., AND HWU, W. W. Program Optimization Space Pruning for a Multithreaded GPU. In *Proc. the Intl. Symp. Code Gen. and Opt.* (2008), pp. 195–204.

[23] SCHAA, D., AND KAELI, D. Exploring the Multiple-GPU Design Space. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 1–12.

[24] SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. Scan Primitives for GPU Computing. In *Proc. ACM Symp. Graphics Hardware* (2007), pp. 97– 106.

[25] VALIANT, L. G. A Bridging Model for Parallel Computation. *Comm. ACM 33*, 8 (1990), 103 – 111.

[26] VINEET, V., AND NARAYANAN, P. J. CUDA Cuts: Fast Graph Cuts on the GPU. In *Proceedings of the CVPR Workshop on Visual Computer Vision on GPUs* (2008).

[27] VIOLA, I., KANITSAR, A., AND GROLLER, E. Hardware-Based Nonlinear Filtering and Segmentation using High-Level Shading Languages. In *Proc. IEEE Visualization* (2003), pp. 309–316.

Appendix

Algorithm 1 Global Memory Benchmark

Input: Number of elements N, $stride$, $offset$, and array A in global memory

1: Calculate no. of elements per thread, N_{thread}
2: Calculate this thread's data range using $stride$ and $offset$
3: **while** $index$ is in range **do**
4: Read $A[index]$ in register R
5: Increment R and store back in $A[index]$;
6: $index = index + stride$
7: **end while**

Algorithm 2 Shared Memory Benchmark

Input: Number of elements N, $stride$, $offset$, $bank$, array A in global memory and array B in shared memory

1: Calculate no. of elements per thread, N_{thread}
2: Calculate this thread's data range using $stride$ and $offset$
3: **while** $index$ is in range **do**
4: **for** $i = 0$ to 10000 **do**
5: Read $A[index]$ and store in
6: $B[ID_{thread} \times bank \pmod{size_{block}}]$
7: **end for**
8: **end while**