A Parallel Algorithm Development Model for the GPU Architecture

J. Steven Kirtzic, **Ovidiu Daescu**
Department of Computer Science
University of Texas at Dallas
Richardson, TX USA
{jsk061000, daescu}@utdallas.edu

Abstract— *Parallel computing has been in use for decades, and throughout many researchers have sought to define a model for algorithm design for such a platform. Valiant developed a model for parallel computing, which was later extended to later include multi-core processors, but it still may not be best suited for the unique GPU architecture. With the current advances in high performance computing, it is easy to see the role that GPUs can play, and even easier to see the need for a model for GPU algorithm development. Here we propose a parallel GPU model which offers both a general design and a fine-grained approach, intended to accommodate nearly any GPU architecture. We show how our model can result in significant increases in performance when algorithms are designed based on its principles.*

Keywords: GPU, parallel processing, algorithm design

1. Introduction

The rapid advancement of the Graphics Processing Unit, or *GPU*, over the last few years has opened up a new world of possibilities for high-speed computation, ranging from biomedical to computer vision applications. Recent examples include [1], [2], and [3]. However, the GPU architecture is unlike that of any other, and designing algorithms to fully harness the capabilities of a GPU is not an easy task, especially when one considers the advantages and disadvantages of the various resources that a GPU has available to it. In this paper we introduce a parallel algorithm design model for the GPU architecture which addresses these issues. In Section 2 we discuss related work; in Section 3 we present a brief overview of the GPU architecture, focusing on NVIDIA's CUDA architecture; in Section 4 we present the model in its entirety; in Section 5 we illustrate the use of our model as we apply it to template and shape matching algorithms; in Section 6 we discuss the results of our model as applied to these template and shape matching algorithms; and finally in Section 7 we conclude and remark on future work.

1.1 Contribution

We believe that our main contribution with this work is to provide an easily accessible parallel algorithm design model for the GPU architecture. Our model addresses the limitations of other parallel models in that it accounts for the unique architecture of the GPU, in particular the various types of memory that the GPU possesses and their individual attributes. Our model is also designed to include single or multi-core CPUs as part of the system, if the designer chooses to do so. Finally, our model is intended to be easily accessible for a wide variety of reseachers from all scientific fields interested in GPU algorithm design, ranging from the novice to the experienced.

2. Related work

We present the following parallel model designs in succession to demonstrate the evolution of our Parallel GPU Model (PGM) and give it proper context.

2.1 The PRAM model

The PRAM model is generally regarded as one of the original parallel algorithm design models. The main shortcoming of the PRAM model lies in its unrealistic assumptions of zero communication overhead and instruction-level synchronization. Another drawback of with the PRAM model is that the time complexity of a PRAM algorithm is often expressed in big-O notation, which is often misleading because the machine size n is usually small in existing parallel computers. Consequently, the PRAM model is generally not used as a machine model for real-life parallel computers.

2.2 The BSP model

The BSP, or bulk-synchronous parallel model, was proposed by Leslie Valiant [4] to overcome the limitations of the PRAM model [5], while maintaining its simplicity. In the BSP model, a BSP computer consists of a set of n processor/memory pairs (nodes) that are interconnected by a communication network. The BPS model is *Multiple Instruction Multiple Data* (MIMD) in nature, and uses the concept of a *superstep*, which is comprised of a computation step, a communication step, and a synchronization step. The BSP model is also variable grained, loosely synchronous, has non-zero overhead, and uses message passing or shared variables for communication.

The program executes as a strict sequence of supersteps. In each superstep, a process executes the computation operations in at most w cycles, a communication operation that

takes gh cycles, and a barrier synchronization that takes l cycles. Note that in the communication overhead gh, g is the proportional coefficient for realizing a h relation. The value of g is platform-dependent, but independent of the communication pattern. In other words, gh is the time that it takes to execute the most time-consuming h relation.

Within a superstep, each computation operation uses only data in its local memory. This data is put into the local memory, either at the program start-up time or by the communication operations of previous supersteps. Therefore, the communication operations of a process are independent of other processes.

The BSP model is more realistic than the PRAM model because it accounts for all overheads except for the parallelism overhead for process management. The time for a superstep is estimated by the sum

$$w + gh + l \tag{1}$$

This model is highly regarded and has formed the basis for other parallel models, such as the parallel phase model [5], which we will briefly discuss next. However, its generality is its shortcoming when one attempts to apply it to more specific architectures, such as that of the GPU. Valiant recently extended his model to include multi-core CPUs [6]. While this model is much more akin to the architectural nature of the GPU, it still does not take into consideration the complexities of the typical GPU architecture, in particular the various types of memory, which as we will demonstrate in later sections have a tremendous impact on the performance of a given GPU algorithm.

2.3 The parallel phase model

Kai Hwang and Zhiwei Xu [7] proposed a phase parallel model for parallel computation that is further refined from the above two abstract models. This model is similar to the BSP model with the following distinctions: a parallel program is executed as a sequence of phases: the parallelism phase, the computation phase, and the interaction phase. The total execution time of the superstep on n processors is expressed by

$$\begin{aligned} T_n &= T_{comp} + T_{interact} + T_{par} \\ &= (w + \sigma\sqrt{2logn})t_f + t_0(n) + \alpha * w * t_c(n) + t_p(n) \end{aligned} \tag{2}$$

where w is the number of cycles, as with the BSP model, α is the *communication-to-communication ratio* (CCR) of each superstep, and t_f is the average time to execute a flop by a processor.

Improved from the PRAM and the BSP models, the phase parallel model is closer to covering real machine/program behavior. In this model, all types of overheads are accounted for, as shown in Eq. (2): the load imbalance overheads, the interaction overhead (t_0 and t_c terms), and the parallelism overhead (t_p term).

While these models represent the evolution of parallel algorithm design in general terms, they are limited in scope as they ultimately fall short when applied to the unique architecture of the modern GPU. The need for a model suited to this architecture was vocalized in a paper from MIT [8] in which the authors identify that official documentation for CUDA from NVIDIA was rather sparse, the forums required a lot of searching to find an answer to a particular problem, and the trade-offs between various programming options were difficult to discern. We attempt to address these issues by providing a model which was designed to not only include the more general models identified above, but to also take into consideration the unique nature of the GPU architecture, as it differs considerably from the CPU architecture.

3. GPU architecture

In this paper we will often refer to the machine containing the GPU as the “host" and the GPU itself as the “device”. The NVIDIA GeForce 8800 series is an example of a typical GPGPU (General Purpose GPU) device, which utilizes NVIDIA’s CUDA (*Compute Unified Device Architecture* GPU design. The GeForce 8800 contains 16 multiprocessors, each containing 8 semi-independent cores for a total of 128 processing units. Each of the 128 processors can run as many as 96 threads concurrently, for a maximum of 12,288 threads executing in parallel. The computing model is SIMD (*Single Instruction Multiple Data*), and the memory model is NUMA (*Non-Uniform Memory Access*) with a semi-shared address space. This stands in contrast to a modern CPU, which is typically either SISD (*Single Instruction Single Data*) or MIMD, in the case of a multi-processor or multi-core machine. Additionally, from the perspective of the programmer, all memory is explicitly shared (in multi-threading environments) or explicitly separate (in multi-processing environments) on a desktop machine.

3.1 GPU instruction throughput versus memory access

The GPU architecture is much more optimized for performing calculations than for memory accesses. Therefore, considering the multiple types of memory that the GPU architecture typically includes, it is important to keep this in mind when accessing these types of memory, particularly the slower, off-chip ones such as the GPU’s global and the host’s main memory. The most costly memory access is by far the host-to-device (CPU to GPU) data transfer, and reducing that transfer can have a tremendous impact on the overall performance of any algorithm that is implemented in part or fully on a GPU.

As an example of our research, we present the case of a typical Full Search Method of template matching, which is otherwise known as a “brute force" method. A naïve GPU

implementation of this algorithm is relatively easy, as the underlying architecture (such as CUDA) will handle most of the scheduling, thread allocation, memory management, etc. for you. In this case, a naïve, straightforward GPU implementation should run in *O(mn/p + log m)* time, where p is the number of processors, assuming that $1 \ll n$. We present the results of the implementation of this naïve GPU algorithm using several different types of GPU memory (discussed below) versus the serial implementation in Table 1.

Table 1: Run time for Full Search Method template matching for a 512x512 image and a 64x64 template. Times are in ms.

	Run Time	Copy Time
CPU	23290	N/A
GPU	3042	217.7
GPU Shared Memory	200.68	217.7
GPU Texture Memory	107.38	2.361

Table 2: Average results over 1000 trials of basic CUDA memory operations. The first column refers to the amount of data used for this experiment, in bytes. "malloc" and "malloc 2D" refer to allocating an array and a byte aligned 2 dimensional array on the GPU, respectively. "copy" and "copy 2D" refer to copying data from the CPU's global memory to the GPU's global memory. All times are in ms.

size	malloc	copy	malloc 2D	copy 2D
$4 * 10^3$	0.067567	0.005253	0.116700	0.014929
$4 * 10^5$	0.118616	0.291486	0.122187	0.296680
$4 * 10^6$	0.141160	2.576290	0.180513	2.713126
$4 * 10^7$	0.241793	23.344471	0.629537	24.801236

Given the considerable differences in architecture between the GPU and CPU, one can see that the ratio of overall run-times of the CPU to naïve GPU implementation (which we define as "speedup", *S*) is only 23290/3042 = 7.66. Given the number of processing cores p in our GPU is 128, this is clearly not an optimal solution, as it yields an efficiency of .060 (we define ''efficiency" as *E* = *S/p*). The majority of this is due to communication overhead (data transfer), as global memory on the GPU is uncached. Experimentation confirms that the instruction throughput is only .034, indicating that 96.6%≈97% of the total run time was due to host-device data transfer.

In addition to the host-device memory read/write, there are several other types of memory that a GPU may have and access, either on-chip or as part of the graphics card, including (in order of typical size) global memory, L2 cache memory, texture memory, shared (local) memory, and the processor registers. The type of memory that a programmer uses for a particular operation depends upon the size and nature of the data structures to be used, whether or not these data structures can be broken up (and if so how), and whether they are read/write data or simply read-only. In Table 2 we present the results of our experimentation with host-device data transfer times for different sizes of data to illustrate the importance of proper data partitioning.

4. Parallel Algorithm Design Procedure for GPUs

As discussed earlier, Valiant's multi-core parallel algorithm model falls short when one attempts to apply it to many-core GPUs. In designing our Parallel GPU Model, we opted to refer back to Valiant's original BSP model as a basis, and build out our model from there.

4.1 General model

GPU Superstep:

$$max_{i=1}^{p} w_i + max_{i=1}^{p} (h_i * g) + l \tag{3}$$

where p = number of processors (cores) on the GPU
w_i = cost of local computation in process *i*
h_i = number of messages sent and received and/or variables accessed by process *i*
l = cost of synchronization
g = message speed (bandwidth)
Algorithm total cost:

$$\begin{aligned} \mathrm{C}_{GPU} &= \sum_{i=1}^{S} superstep_i \\ &= W + H * g + S * l \\ &= \sum_{s=1}^{S} w * s + (\sum_{s=1}^{S} h_s) g + S * l \end{aligned} \tag{4}$$

where S = number of supersteps
W = total cost of local computations in all processes
H = total number of messages sent and received and/or variables accessed by all processes

CPU Superstep:

The CPU component of this general model is very similar to the GPU component above, with the exception that the variables apply to the CPU (i.e. p applies to the number of CPU cores, processes are executed on the CPU cores, etc.)

Total

$$C_{total} = \sum_{i=1}^{n} C_{GPU_i} + \sum_{i=1}^{n} C_{CPU_i} + (\sum_{i=1}^{m} T_i) b \tag{5}$$

where C = cost
n = number of algorithms or parts of algorithms executed
m = number of data transfers between CPU and GPU (usually an even number)
T = one-way data transfer
b = CPU to GPU data transfer bandwidth

In the simplest, single algorithm situation, this can be represented as:

$$C_{total} = C_{GPU} + C_{CPU} + 2 * T * b \quad (6)$$

Reducing the number of Ts along with the size of T(the amount of data transferred) are two coarse-grained methods of reducing an algorithm's overall run time and thereby increasing performance.

4.2 Fine-grained model for parallel GPU algorithm design

If the algorithm designer is experienced in parallel algorithm design principles, they can typically skip to Step 3 below, otherwise they should continue with the following steps.

Step 1: Once a serial implementation of a given algorithm has been either acquired or originally designed, the first consideration is what types of instructions (operations) are to be performed on the given data set or sets of the algorithm. This will help in determining data dependency (as discussed below), as well as to help determine what operations must be performed on which processor.

Step 2: The goal is to reduce the total data transfer time as much as possible, meaning reducing the amount of data that is transferred back and forth between the host and device. Furthermore, this round trip may be performed multiple times for one algorithm depending on the structure of the algorithm and/or the size of the data sets. The various types of GPU memory are of varying sizes, but are all generally very small compared to modern host main memory. Therefore, in some parallel applications a data set may have to be broken up and sent to the GPU for computation through several round-trips. Consequently, as discussed briefly in Section 3.1, sending large amounts of data to and from the GPU unnecessarily may lead to a longer run time than the original serial algorithm (especially if it is optimized and/or implemented on a multi-core CPU).

With this type of trade-off of data and operations between the CPU and GPU, the most important aspect in determining what should be transferred to the GPU is *data dependency*. This essentially refers to identifying what operations require the data that is the result of previous operations. If an operation must wait for the resulting data from a previous operation, then these operations must be performed serially. They can still be performed serially on the GPU, but that would defeat the purpose of using the GPU and would require unnecessary data transfers. Furthermore, in most cases the CPU would be able to complete these serial operations faster than a GPU would, even without taking into consideration the costly host-device transfer time.

At this point, more experienced parallel GPU algorithm designers can proceed to Step 4 where we discuss optimizations specific to the GPU architecture. Otherwise it is recommended to take the following step of designing and implementing a naïve parallel algorithm before continuing to the optimization step.

Step 3: Implementing a naïve parallel GPU algorithm is a relatively straight-forward task. Most GPU architectures, such as CUDA, include a thread scheduler which will automatically distribute computations to threads and handle other high level functions of the GPU for you. What this results in is a simple port of a serial algorithm to a GPU with little or no consideration for the various aspects of a GPU's architecture that can be leveraged to create an optimal parallel algorithm. While a naïve parallel implementation can be accomplished rather quickly and easily resulting in a notable speedup of the algorithm's performance, this speedup will not be as great as it could be when the GPU architecture optimizations are performed, as discussed below.

Once the data dependencies within the algorithm have been identified, the designer is then able to break the up algorithm into the various parts that have to be done in serial and the parts that can be done in parallel. From that point, the designer can implement the serial parts on the CPU using typical CPU code (i.e. C/C++, Java, etc.) and implement the parallel parts using GPU code (i.e., C for CUDA, Brook+, etc.). With architectures such as CUDA, you can implement the CPU and GPU code in the same program, with the GPU code written simply as individual kernels that are called from within the CPU code, which simplifies the writing of the code a great deal. Also, recently many CPU and GPU manufacturers have adopted a language known as OpenCL, which allows algorithms to be implemented in a single language that can run on both the CPU and GPU, eliminating the need for separate languages for each architecture [9].

As far as GPU memory manipulation goes, with a naïve implementation it is usually easy to simply load the data set (or as much as possible at one time) into the GPU's global memory. The GPU will handle transferring the data from the host's RAM into the GPU's memory whenever a kernel is invoked. The designer just specifies, in the case of CUDA for example, which memory type is being used for which kernel when the kernel is defined in the code. The global memory, while being the largest type of memory and read-write capable, is also the slowest memory. Therefore it is one of the first areas to avoid, if possible, when performing algorithm optimizations, as is described in the next step.

To achieve optimal or near-optimal performance, we must take into consideration the unique architecture of the GPU and exploit this architecture to its fullest. It may take even the most experienced GPU algorithm designer several attempts to achieve optimal results, as often time optimality is best determined through experimentation. But careful analysis of the algorithm along with the GPU architecture can help to greatly reduce the need for experimentation to achieve optimality.

One basic yet effective step toward optimality is to eliminate unnecessary computations. As discussed above, naïve brute force parallel algorithms simply perform the same computation on an entire data set in one step (or several steps depending upon the size of the data set and the number of available threads). Our implementation of a naïve template matching algorithm searched the entire image for a match to the template, which required the transfer of the entire image data from the host's main memory to the GPU's global memory, a very costly operation. However, by redesigning our algorithm to first perform a "pruning step", we were able to reduce the overall runtime by as much as 99%, depending upon the amount of noise in the original image [12]. This is an example of placing "smart" bounds on the dataset to greatly reduce the amount of data that has to be transferred from the host to the device.

The next step in creating optimal parallel algorithms for the GPU is to determine what type and size of data structures your algorithm will use. These considerations are closely associated with what types of memory the algorithm will utilize, both within and outside of the GPU. As discussed in Section 3.1, the GPU architecture is unique in its design and varies considerably from that of a CPU. Indeed, it's fair to say that a GPU is analogous to being "a computer within a computer". Thus the various types of memory that a GPU contains and has access to certainly complicates considerations when designing parallel GPU algorithms. Following, we will discuss the various types of CPU/GPU accessible memory and their advantages/disadvantages.

4.2.1 GPU memory considerations

We denote a system's RAM as M, and note that it has the following attributes: it is read/write capable, is the largest sized memory overall (typically in the order of GB by current standards), its transfer speed (host to device) is the slowest by far, and it is not directly accessible by kernel threads. Global memory, which we denote with G has the following attributes: it is read/write capable, is the largest GPU (on card) memory, its transfer speed is the slowest for a given GPU and graphics card, and it is accessible by all threads. The L2 cache, denoted by L, is an example of a high capacity, high speed component that may or may not be available on a particular GPU, depending upon the model that one is using, such as in the case of the Fermi architecture (see [10]). The texture memory, which we denote as x, has the following attributes: it is read only, is smaller than global memory, but larger than shared memory, it is much faster than global memory but not as fast as shared memory, and is accessible by all kernel threads. The next type of memory is the shared local memory/L1 cache (in the cases where shared local memory also includes an L1 cache [10]). Shared local memory has the following attributes: it is read/write capable; is much smaller than texture memory but larger than the registers; and is somewhat faster than texture

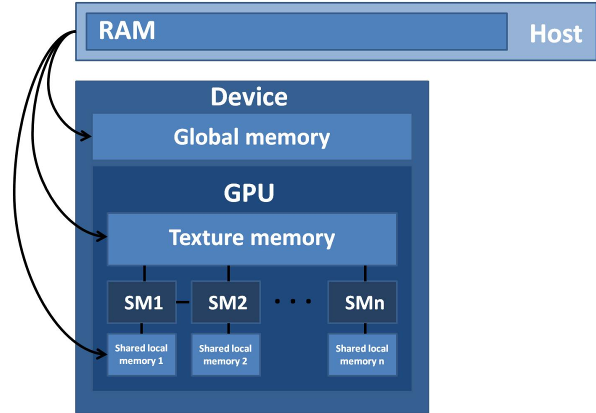

Fig. 1: The host(CPU)-device(GPU) memory hierarchy available to programmers (note that registers are excluded as they are not directly addressable by GPU programmers). We can see how the GPU must go through the host to load data from the RAM into the GPU's global, texture, or shared memories, the latter of which can communicate through SMs (Streaming Multiprocessors).

memory, but is accessible only by threads on a Streaming Multiprocessor (SM). Finally, we denote the register usage by$r$, where registers have the following attributes: they are read/write capable, they are the smallest and fastest of all memory types, and they are typically one register per core. We illustrate the various types of memory available to the GPU programmer with their relative sizes in Figure 1.

With this in mind, however, we will omit the following terms from the final PGM for the following reasons: the RAM transfer time (i.e. host-to-device) is accounted for by the term Tb in Equation (1). Further, we consider the fact that the registers are simply used by the cores as "scratch pads" to temporarily hold the data to be used by the processors and thus are not able to be directly manipulated by the programmer. Therefore, with the exception of bank conflict concerns, the register use should not be of consequence to the designer, and can then be eliminated from consideration.

Naturally, we wish to use the fastest type of memory available in all occasions. However, due to limitations imposed by the size and nature (i.e. read only or read/write) of the data structures that we use for a given algorithm, along with the sizes of the various types of memory as discussed above, a designer may choose different types of memory for various data structures. For example, with our template matching algorithm, we chose to use the texture memory to load our initial query image into, due to the fact that the texture memory is large, fast, and the image does not require write capabilities [12].

Obviously, with the goal being to take as much advantage of the fastest types of memory as possible, a designer's data structures may need to be altered to adhere to the size

limitations of certain forms of memory. As an example, if image data can be streamed into the shared memory blocks at a faster rate than transferring the entire image into global or even texture memory, (without disrupting the computation being performed in those blocks) then using the faster shared local memory is preferable.

Once the algorithm designer has an understanding of the various types of memory that the GPU provides, they can proceed to design the data structures that their parallel algorithm utilizes accordingly. This includes deciding how to break up the data itself, so that the size and the nature of the data structures allow them to be grouped into logical blocks that can be mapped into thread blocks of one, two, or three dimensions, where the threads can maximize their intercommunication through variable sharing within the SM's shared local memory. With our GPU template matching algorithm (discussed in Section 5), we demonstrate the advantages of not only breaking up of the data into logical blocks, (or "strips" in this case) but also the smart use of the GPU's texture memory instead of the GPU global memory, with the former being much faster.

It is important at this point to reinforce that the naïve parallel implementation will typically be subject to the underlying system's automatic (typically non-optimal) scheduling system, which will typically distribute threads on a first-come-first-serve basis, which is generally not optimal.

Another important technique toward achieving optimality with a given parallel GPU algorithm is identifying and reducing (or even eliminating) algorithm execution bottlenecks. As we have discussed above, the execution bottlenecks primarily are memory transfers, in particular the host to device transfer. This issue is being addressed with the new generation of CUDA architecture (Fermi [10]), but for the present the average algorithm designer and implementer dealing with commodity GPUs needs to consider the massive host-device bottleneck.

Above we discussed several ways to reduce the amount of data transfer between the host and device, as well as ways to design your data structures to fully take advantage of common GPU architectures. Following we shall formally define our Parallel GPU Model based upon the above considerations.

Previously, we defined a GPU operation in the following manner:

$$\begin{aligned} C_{GPU} &= \sum_{i=1}^{S} \text{superstep}_i \\ &= W + H * g + S * l \\ &= \sum_{s=1}^{S} w_s + (\sum_{s=1}^{S} h_s) g + S * l \end{aligned} \tag{7}$$

where S = number of supersteps, W = total cost of local computations in all processes, H = total number of messages sent and received and/or variables accessed by all processes, w_s = cost of local computation in process s, h_s = number of messages sent and received and/or variables accessed by process s, l = cost of synchronization, and g = message speed (bandwidth)

Generally parallel algorithms work toward computing a result from a large number of simpler calculations performed in parallel on the various processors/cores in the system. This requires a reduction step, which as discussed above typically runs in O(log *m)* time. We represent this step as *R* and add it to Equation (7), which gives us:

$$\sum_{s=1}^{S} w_s + (\sum_{s=1}^{S} h_s) g + S * l + R \tag{8}$$

The PGM is essentially an extension/adaptation of existing parallel algorithm models, including the PRAM, BSP, and Parallel Phase Model. However, our model is focused on the GPU architecture, which requires the redefinition of certain terms from the original BSP/Parallel Phase Model. In our model, we equate a BSP/Parallel Phase Model superstep with the execution of a GPU "kernel", which is essentially a GPU function or method which handles the importing of the data set, the computations to be performed on said data set, and the exporting of the resulting data to the CPU (i. e. RAM). Therefore we shall now denote a kernel/superstep as k, with the total number of kernels executed as K. This gives us an updated version of Equation (8) as:

$$\sum_{k=1}^{K} w_k + (\sum_{k=1}^{K} h_k) g + S * l + R \tag{9}$$

4.2.2 Message/variable passing

The term $(\sum_{k=1}^{K} h_k)g$, which describes the total number of messages and/or variables transmitted multiplied by the bandwidth of the transfer medium, is perhaps the most important of the terms and deserves more consideration. As we expand this term to include the various types of memory that a GPU can read from and write to, we get

$$\begin{aligned} (\sum_{k=1}^{K} h_k) g =& (\sum_{a=1}^{A} h_a) G + (\sum_{b=1}^{B} h_b) L \\ &+ (\sum_{c=1}^{C} h_c) x + (\sum_{d=1}^{D} h_d) y \end{aligned} \tag{10}$$

where a = an individual global memory read/write
A = the total global memory reads/writes
b = an individual L2 cache memory read/write
B = the total L2 cache reads/writes
c = an individual texture memory read/write
C = the total texture memory reads/writes
d = an individual shared memory read/write
D = the total shared memory reads/writes

When we substitute this more accurate representation of memory reads/writes into Equation (2), we get:

$$\sum_{k=1}^{K} w_k + (\sum_{a=1}^{A} h_a)G + (\sum_{b=1}^{B} h_b)L + (\sum_{c=1}^{C} h_c)x + (\sum_{d=1}^{D} h_d)y) + S * l + R \quad (11)$$

It should be noted that with memory transfers we do not take into consideration the size of a word for the particular system, so we are utilizing the *uniform cost criterion*, as is commonly done with parallel algorithm design and analysis [11]. Also, based upon the ratios of run times between the global, texture, and shared local memories, we can calculate coefficients to represent the approximate cost of using each type of memory and add these coefficients to Equation (5). Normalized with respect to global memory we get ratios of 0.035 and 0.066 for the texture and shared memories, respectively. Note that these two values are roughly in a ratio of 1 to 2 in relation to each other. Applying this to Equation (5) then gives us the following:

$$\sum_{k=1}^{K} w_k + (\sum_{a=1}^{A} h_a)G + (\sum_{b=1}^{B} h_b)L + 0.035(\sum_{c=1}^{C} h_c)x + 0.066(\sum_{d=1}^{D} h_d)y + S * l + R \quad (12)$$

Which can be generalized to:

$$\sum_{k=1}^{K} w_k + (\sum_{a=1}^{A} h_a)G + (\sum_{b=1}^{B} h_b)L + \frac{1}{32}(\sum_{c=1}^{C} h_c)x + \frac{1}{16}(\sum_{d=1}^{D} h_d)y + S * l + R \quad (13)$$

Thus, with the above equation we can see the relative costs of using various types of memory addressable by the GPU programmer. This model is intended to be applicable to most, if not all GPU architectures: indeed, if a particular architectural feature is not available with the model of GPU being employed (i.e. L2 cache memory), then that term is simply zeroed out or removed from the above equation.

Step 5:The last step of this Parallel GPU Model design procedure is more advanced and involves a more intimate knowledge of the specifics of the architecture of the particular GPU the algorithm is designed for. It should be noted that this step is not necessary to achieve near-optimality, as that can typically be achieved by adhering to the above steps. However, for the designer desiring as much optimality as possible, they should identify and understand several physical aspects of the particular device. These physical aspects include the following: the number of processors/cores that the GPU has; the number of SMs the GPU has; the number of processors/cores per SM; the amount of global, texture, and shared local memory the GPU has; the type and speed of the connection between the host and the device (i.e. PCI, PCIe, etc.); the availability and size of the L1 and the L2 caches (as with Fermi GPUs); whether or not the particular GPU architecture supports IEEE 754-2008 (which includes full double precision support); the nature of the GPU's warp scheduler (i. e. whether or not it is a single or dual warp scheduler); whether or not the device has Error Correcting Code (ECC) memory support; whether the device is 32 or 64 bit-based; and what programming languages the device supports, such as C/C++ for CUDA, OpenCL, etc. (and to what degree).

By knowing hardware-specific details, such as the number of processors/cores in the particular GPU, we can augment our abbreviated PGM with the above substitution, which yields the following version of the PGM:

$$\sum_{k=1}^{K} \frac{mn}{p} + (\sum_{k=1}^{K} h_k)g + S * l + R \quad (14)$$

where *mn/p* = w_k
m = number of computations to perform
n = number of data elements
p = number of processors/cores

5. Applications of the PGM

We first applied our GPM to the field of template matching in [12]. Here we developed a GPU-accelerated template matching algorithm from the ground up based upon the PGM. Template matching essentially involves searching an image *I* attempting to find the match for a template *x* among all possible candidates y_i in a sort of "sliding window" fashion. The searches are independent of each other and therefore are highly subject to parallel processing on the GPU. Furthermore, we employed a pruning step to eliminate unnecessary data transfer to the GPU, as discussed in Step 3 of Section 4 above.

In our second application of the PGM, we chose to apply it to a shape matching algorithm that we had developed [13] previously. In this case, a 3D shape is given a unique "signature" which is calculated by computing the distances between each vertex of the shape and every other vertex, as long as that vertex is "visible" to the original vertex. The result is a large number of calculations which are then formulated into a histogram, which then forms the shape's unique signature. Obviously, these calculations are completely independent of each other and therefore this algorithm is also a very good candidate for parallel processing on the GPU.

6. Results and discussion

The experimental design for our template matching algorithm consisted of averaging the results of running our algorithm over a number of trials with a variety of images of different sizes and resolutions, which yielded the following performance results: when comparing the performance of our template matching algorithm to the Full Search Method discussed earlier on small images (512x512) at zero to low noise levels, our algorithm has better performance than the Full Search Method. When comparing our algorithm's performance to that of a standard brute-force Full Search Method implemented serially on the CPU on medium to large images one can see the tremendous performance increase of our algorithm. With an image size of 1024x1024 and a template size of 256x256, our algorithm experiences a 8700x performance increase over the Full Search Method. Further, when we implemented the Full Search Method in a naïve parallel manner on the GPU, our optimized algorithm performed 39x faster.

Similarly we observed a considerable speedup in run-time in our shape matching algorithm when applying the PGM to it. Serially, this algorithm has a run-time of:

$$O(n^2 \ \lambda(n) log(n/\epsilon)/\epsilon^4 + n^2 \ log(np) \ log(n \ log \ p)$$

However, with the application of the PGM, we observe the following: if the number of data items equals *n* and the number of processors equals *p*, then the total computation time for the above shape matching algorithm is:

$$n/p + logn \qquad (15)$$

where *log n* is the reduction step. This is assuming that at each timestep a processor *p* is calculating the distance from a query point to another point. The reduction step results in the shortest distance from the query point to the signature point. Therefore, this algorithm could perform in near-linear time, depending upon the number processors in the parallel system.

In comparison with other parallel design models, we observe the following: the PRAM model, while being a fundamental and an "all-encompassing" parallel design model, is rather inadequate when applied to modern GPGPU design, due to its generality. Valiant's BSP model is a MIMD model consisting of node/memory pairs interconnected through a network. This is not very analogous to modern GPU architectures. The parallel phase model attempts to make up for this shortcoming by accounting for all overhead costs, but is still based on a model which does not account for the various types of memory that the GPU possesses, nor their individual advantages and disadvantages. Valiant's more recent multi-core model is much more akin to the many-core GPU architecture, however it still falls short when one considers the various types of GPU memory, including their attributes and design concerns. We believe that our model is well-suited to the GPU architecture by accounting for all possible types of memory and their associated costs that a GPU can access, both with current commodity GPUs and new, more advanced architectures. Furthermore, we provide a thorough analysis of the various considerations that one must keep in mind when designing algorithms for any GPU architecture, and we believe that our model provides an opportunity to do so that other models don't.

7. Conclusion and future work

Overall, we believe that our parallel GPU method is very effective in allowing parallel GPU algorithm designers, ranging from the novice to the expert, to design and implement optimal (or nearly optimal) algorithms that take advantage of the GPU architecture. We noted that while previous models have been adequate for general parallel architectures (which can vary considerably), they fall short when addressing the unique architecture of GPUs. Indeed, the degree of performance that can be achieved with the application of other parallel models, such as the BSP and phase parallel model, is not optimal for the GPU architecture, and we showed how the PGM can achieve a greater degree of optimality. Therefore, depending upon the degree of optimality desired, our model seems superior to other existing parallel models when applied to the GPU architecture.

Future work with our model will include applying it to simulations involving radiation therapy, such as Volume Modulated Arc Therapy (VMAT) and Intensity Modulated Radiation Therapy (IMRT). In such simulations, various computations need to be made in real-time or near real-time and the use of GPUs would certainly be a great advantage. Also, more practical experience with a Fermi GPU would allow for the implementation of various algorithms to quantitatively measure the speedup that a particular architecture allows over other parallel and serial architectures. Finally, the development of software which employs an "automated" version of this model would make it accessible to even more researchers by aiding them in making more informed with their algorithm designs, thereby producing even more optimal results.

References

[1] D. Qiu, S. May, and A. Nüchter, "GPU-accelerated nearest neighbor search for 3d registration," *Computer Vision Systems*, pp. 194–203, 2009.

[2] P. Noël, A. Walczak, K. Hoffmann, J. Xu, J. Corso, and S. Schafer, "Clinical evaluation of GPU-based cone beam computed tomography," *Proc. of High-Performance Medical Image Computing and Computer-Aided Intervention (HP-MICCAI)*, 2008.

[3] J. Huang, S. Ponce, S. Park, Y. Cao, and F. Quek, "GPU-accelerated computation for robust motion tracking using the CUDA framework," in *Visual Information Engineering, 2008. VIE 2008. 5th International Conference on*. IET, 2008, pp. 437–442.

[4] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990. [Online]. Available: http://portal.acm.org/citation.cfm?id=79181

[5] K. Hwang and Z. Xu, *Scalable parallel computing: technology, architecture, programming*. WCB/McGraw-Hill, 1998.
[6] L. Valiant, "A bridging model for multi-core computing," *Journal of Computer and System Sciences*, vol. 77, no. 1, pp. 154–166, 2011.
[7] Z. Xu and K. Hwang, "Early prediction of mpp performance: the sp2, t3d, and paragon experiences," *Parallel Computing*, vol. 22, no. 7, pp. 917–942, 1996.
[8] Massachusetts Institute of Technology, "IAP09 CUDA@MIT 6.963," MIT, 2009. [Online]. Available: http://sites.google.com/site/cudaiap2009/home
[9] "Opencl programming guide for the cuda architecture v2.3."
[10] NVIDIA Corp., "NVIDIA's next generation CUDA compute architecture: Fermi," Sept. 2009. [Online]. Available: http://www.nvidia.com/object/fermi_architecture.html
[11] J. Jaja, *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., 1992.
[12] R. Anderson, J. Kirtzic, and O. Daescu, "Applying parallel design techniques to template matching with gpus," in *High Performance Computing for Computational Science–VECPAR 2010: 9th International Conference, Berkeley, CA, USA, June 22-25, 2010, Revised, Selected Papers*, vol. 6449. Springer-Verlag New York Inc, 2011, p. 456.
[13] Y. K. Cheung and O. Daescu, "Approximate point-to-face shortest paths in ^{3}," *CoRR*, vol. abs/1004.1588, 2010. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1004.htmlabs-1004-1588