GPU Computing

The Ascend of the Coprocessor

Ashraf Al-Suyyagh

GPU Computing – The Ascend of the Coprocessor

By Ashraf Al-Suyyagh
mrsuyyagh@gmail.com

Abstract

GPUs are specialized image synthesis hardware which produce images by operating on descriptions of a scene in the form of triangle matrices specified in high precision floating point format. This was traditionally done through a fixed hardware pipeline embracing parallel design to achieve performance. Recent GPUs replaced fixed pipeline stages by many programmable cores to allow for programming flexibility and offer dynamic load balancing. This high programmability, high precision, extensive parallelism have motivated hundreds of researchers and scientific institutions to develop suitable programming models for this processing power, others to come up with scalable parallel versions of complex algorithms, and some to explore architectural design space improvements and memory models to reduce latency and improve throughput. This research report aims to examine the abstract architectural innovations in modern generations of NVIDIA GPUs, it will also explore the CUDA programming model provided by NVIDIA to facilitate general purpose computing on GPUs, furthermore, performance gains and limitations of GPU computing and energy efficiency of GPUs are discussed.

1. Introduction

Performance growth slowdown of single core processors due to the difficulty of exploiting higher instruction level parallelism, coupled with technological limitations in semiconductor scaling as well as power and thermal challenges has led to a shift in microprocessor design strategies and migration to building multi-core processors. [1] Unfortunately, the full power of these multi-core chips is not yet fully exploited due to the humble numbers of available parallelized commercial software and implicit support for parallel programming is still in its infant stages. Moreover, even the most sophisticated multi-core chips of today are relatively inefficient for high performance computing for a very small amount of the chip is truly devoted for direct calculations, while the rest of the hardware is devoted to architectural innovations targeting single thread performance enhancements. [3]

Meanwhile, commodity GPUs have rapidly evolved in performance, architecture, and programmability offering application potential beyond their primary purpose of graphical processing [2]. In fact, the rate of performance growth of GPUs outpaces Moore's law as applied to traditional CPUs, and the GPU-CPU gap is still widening [4]; however, this performance is not absolute for GPUs only excel in some classes of applications while CPUs in others [2]. In this context, GPUs are attractive for they provide extensive resources, massive parallelism, high arithmetic intensity, memory bandwidth and high-precision platform and most importantly a cost effective solution for high-end computing and scientific applications. GPUs are thus considered "many core processors".

Since GPUs are built from the ground up to support graphical processing, , support from graphical programming models for general purpose computing proved to be inefficient and highly restrictive; therefore the need for a new programming model which balances hardware costs and provides programming convenience by abstracting the complexity of the graphical

hardware had to be developed [1]. Such successful programming model is the NVIDIA CUDA. This report focuses on hardware GPUs and the programming model of NVIDIA following the track of most worldwide research because NVIDIA is a pioneer in exploring the GPU computing space and that it offered its solutions earlier to the market and academia thus gaining wide familiarity and acceptance.

The blazing processing power of modern GPUs and their ubiquitous spread coupled by the introduction of suitable programming models by graphic processors vendors caught the eye of the scientific community to utilize these highly parallel programmable processing engines in scientific research and applications with unmatched results by the most powerful CPUs. Moreover, though the power demands are rated higher than most modern CPUs, GPUs are considered energy efficient considering the little time they are using that energy [5]. This has motivated hundreds of researchers and scientific institutions to come up with scalable parallel versions of complex algorithms.

The primary goal of this report is to present the hardware innovations in GPU hardware organization and the associated programming model which made this revolution in commodity high end computing possible, to examine the performance gains of GPUs over their CPUs counterparts and to present the hardware and software limitations on performance gains of these powerful processing engines.

In this report, section 2 briefly surveys related work in the GPU computing domain. Section 3 presents the internal architecture of different modern generations of NVIDIA graphical processors and outlines design aspects which made the GPU computing possible. In section 4, we present two programming models for GPUs, the traditional graphical model and the CUDA programming model. Section 5 presents performance results and comparisons against CPUs based on various work in literature. Moreover, the implications and limitations of hardware organization and the CUDA programming model on performance are elaborated. Since power has become a major concern in modern day computing, section 6 discusses the energy performance of GPU against CPUs. Finally, the report concludes with a discussion which summarizes the report.

2. Literature Review

Several white papers and IEEE magazines articles explore the design space and internal architecture of several GPUs from NVIDIA. In [6], a formal layout of the operation of GPUs, their pipelines and development is introduced, [12], [13], [14] and [17] extend the previous discussion on NVIDIA hardware solutions for GPU computing, covering the NVIDIA GeForce 6, NVIDIA Tesla and Fermi micro architectures. [3], compares between state of the art CPUs and GPUs and provides an insight of GPU advantages over their counterparts.

In [15], Schaa et al offer a model to predict GPU performance with varying parameters and input sizes, while [1] and [2] analyze actual performance gains of GPUs over single and multithreaded algorithms run on CPUs based on using selected complex algorithms with various communication patterns based on Berkeley's dwarf taxonomy [16]. Moreover, they also explore the programming model and hardware design features impact on performance.

The NVIDIA CUDA programming model is presented in [11] along with some programming examples, [1] has a section which describes successful mapping of the programming model into the hardware. [4] surveys GPU computing hardware and programming models.

In [5] an empirical study on the energy efficiency of GPUs against CPUs is discussed.

3. GPU Hardware Organization and Development

Graphical accelerators have been used for many decades and their numbers have grown tremendously since then driven by the advent of the personal computer. Yet, the interest in using these specialized processors in general purpose computing is only very recent. In this section, we briefly describe the image processing and synthesis technique which has guided the GPU pipeline design, after which we describe the recent developments in GPU architecture and design changes which has revolutionized GPU computing. Moreover, efficient general processing programming assumes that the programmer has sufficient knowledge of the underlying GPU hardware since modern programming models do not fully abstract the hardware.

3D computer graphics assume that everything is made of triangles (specified by their vertices) and objects whatever their complexity is can be described and built from triangles to form geometric primitives and complex shapes. Scenes to be displayed contain geometric primitives, along with descriptions of surface fillings (textures), light sources illuminating the scene, light reflections, and object positioning. 3D graphic systems synthesize images from these scene descriptions. Consequently, all hardware designs were guided by the steps through which an image is rendered, from early scene vertices description to a final pixel version to be sent to the VGA device. [6]

3.1 Early GPU designs

Early graphic hardware designs expressed the image synthesis process as a series of fixed hardwired pipeline implementation with fixed hardware units for vertex transformation and pixel texturing. The pipeline stages perform many single precision floating point operations on the stream of triangles – supplied to the pipeline engine in the form of triangle vertices. The basic pipeline stages included: *Model transformation,* rotating, scaling and translating triangles to a common homogenous coordinate system; *Lightning calculation,* coloring every triangle based on the lights in the scene; *Camera simulation,* projecting the colored stream of triangles on a virtual camera film frame; *Rasterization*, turning descriptions into pixels, since each pixel can be computed independently, this had led to build increasingly parallel set of pipelines; *Texturing*, in which images called textures are draped over the geometry to give the illusion of detail; *Hidden Surfaces calculation*, since objects may obscure other objects in the scene, this stage sorts all triangles from back to front and calculates which pixels are closer to the viewer and update the display accordingly and continuously! [6]

3.2 First Generation Programmable GPUs

As graphic programmers longed to achieve higher levels of detail and realism in their programs, they needed to escape the fixed hardwired implementation. Therefore GPU manufacturers gradually and progressively replaced these fixed hardwired pipeline stages with stage-

specialized user programmable processor units, examples are vertex processors which operate on the vertices of primitives and pixel processors which fill the interior of the primitives, this offered graphics programmers the flexibility to apply a set of programs and algorithms for triangle construction, transformation and filling to achieve the sought realism and not be restricted to the boundaries of the GPU fixed hardwired design [6]. This first generation of programmable GPUs offered high single-precision floating point processing power in comparison to previous designs. Figure 1 shows a hardware model of NVIDIA GeForce 6 series introduced in 2004, an advanced example of the first generation programmable GPU revolution which started three years earlier.

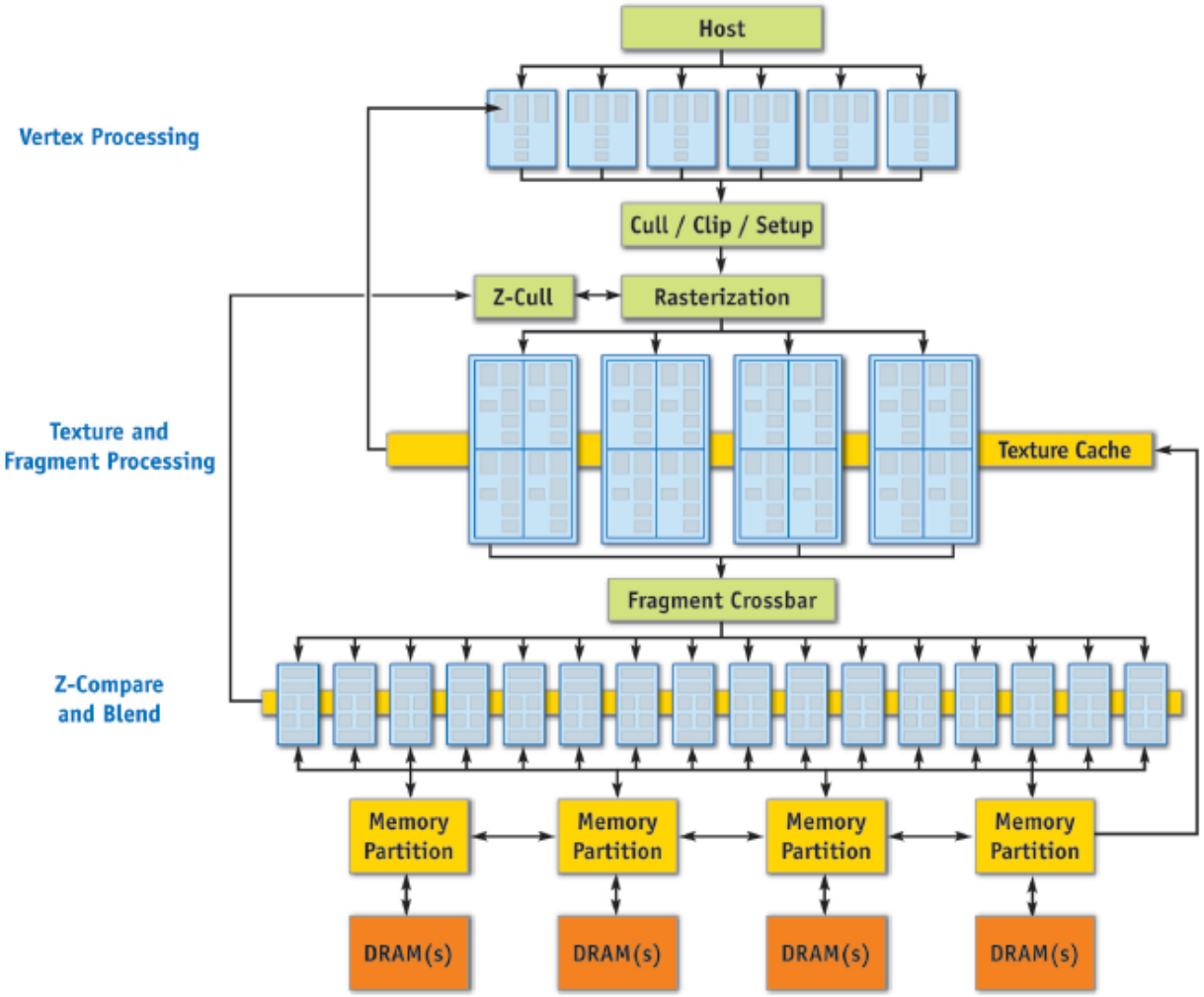

Figure 1. NVIDIA GEFORCE 6 Architecture

3.3 Second Generation Programmable GPUs (1st Generation Unified Device Architecture)

Yet, since 3D graphic engines have to supply images to the display at real time – usually at a rate of 60 times per second [6], and since graphic API generations have allowed scenes to be described in much more complexity and sophistication, a major drawback was exposed: the complexity of the scene often induced high load and congested certain types of processors leaving the other processors idle or underutilized at the same time; an example is when drawing large triangles the vector processors are idle while the pixel processors are busy, meanwhile when the triangles are small, the opposite is true [6][13]. This imbalance would affect the processing and supply rate of images to the display. Therefore, as of 2007, GPU manufacturers introduced the second generation of programmable GPUs (also referred to as the first generation of unified graphics and computing architecture), where in this organization, processor specialization was eliminated and all processors were unified, and called shaders. This unified shader model overcame the imbalance problem by programming the processor to act as a vertex or pixel shader as needed thus offering dynamic load balancing among varying pixel/vertex processing workloads. An example of this new design model is the NVIDIA Tesla microarchitecture which offers scalable processor array [13]. Figure 2 shows the architecture of the NVIDIA GeForce GTX8800 GPU, the first to implement this design. The scalable processor array is divided into eight Texture Processing Clusters (TPC) where each TPC encompasses a couple of Streaming Multiprocessors (SM) each encompassing eight Streaming Processors (SP), which are single precision floating point ALUs, each of which having its own set of 32 registers. [13] A 16kB-16 banks dedicated shared memory is built into each SM, also sometimes referred to as Per-Block Shared Memory (PBSM). Figures 3 shows a detailed internal design of an SM in the Tesla architecture.

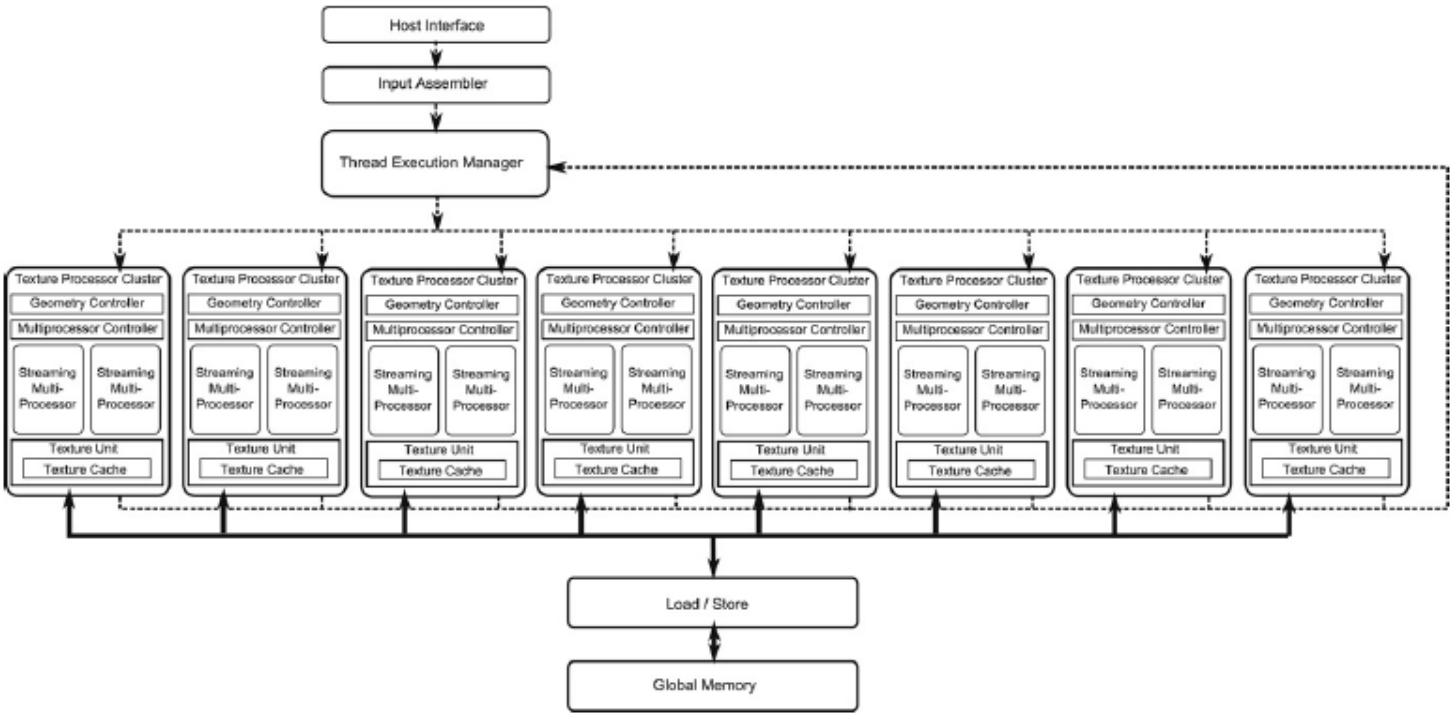

Figure 2. NVIDIA GEFORCE 8800 GTX GPU Organization Based on the Tesla Architecture

3.4 Second Generation Programmable GPUs (2nd Generation Unified Device Architecture)

In 2008, NVIDIA announced its second generation of unified graphics and computing architecture, the GeForce 200 series, in this implementation, NVIDIA increased the number of SMs in each TPC from 2 to 3, and adding two more TPCs roughly doubling the number of SPs from 128 in GeForce GTX8800 to 240 in GeForce GTX280. Most importantly and notably is the migration to use double precision 64-bit floating point units for the first time in GPUs and expanding the bus width to 512 bits offering 65% memory bandwidth increase over the previous generation [17]. These enhancements are also coupled with improved dynamic power management modes which allow the GPU to deliver idle power which is nearly $1/10^{th}$ of the peak performance power.

This presented migration from fixed hardwired graphics pipelines to massively parallel, multithreaded multiprocessor, high precision design was appealing to be used in general purpose computations; specifically is in scientific, engineering and financial applications.

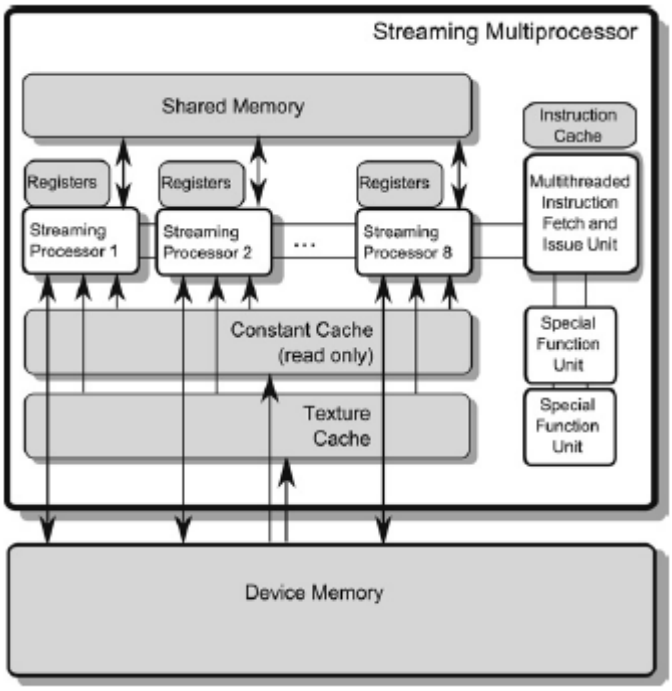

Figure 3. Internal Design Layout of Streaming Microprocessor (SM) in Tesla Architecture

4. GPU Programming Model

GPU programming models depart from conventional CPU sequential counterparts for the latter are not tailored and efficient to run on massively parallel systems, therefore new distinct models had to be developed. Two programming models for GPU computing are available, the graphical model and the streaming model with the prior being extinct and the latter model prevailing in modern development environments.

4.1. Graphical Programming Models

When general purpose GPU computing caught interest, no software support was available to harvest and unleash the power of these processing engines for non-graphical programming. In consequence, applications had to adapt with the limitations of the only available model of the time, the graphical model. Scientific programs had to be mapped into graphic hardware using graphic APIs and shader languages such as DirectX and OpenGL, programmers had to map their data arrays into texture models and think of their program as if it was a graphics program. This approach of reformulating programs in graphic metaphors proved cumbersome, inefficient and highly restrictive leading the way for new models to take place.

4.2. NVIDIA CUDA Programming Model

New programming models based on streaming processor programming models emerged; these models expose the parallelism and communication patterns inherent in the application by structuring data into streams and expressing computation as arithmetic kernels that operate on the streams. [4] NVIDIA, motivated by the desire to use the GPU not only for graphics focused computations but also for general purpose computations, released the Compute Unified Device Architecture (CUDA) programming model and language concurrently with its new Tesla architecture in 2007. This programming model was designed from the base up to support GPGPUs. CUDA programming model extends the C/C++ programming language in such a way to abstract the complex graphic hardware.

In this model, data arrays are partitioned into blocks which are further partitioned into elements on the condition that the blocks can be *independently* computed in parallel and the elements can *cooperatively* be computed in parallel. On this basis, in the CUDA programming model, the programmer writes sequential code which is run on the CPU (called *host*) and executes parallel *kernels* which represent parallel tasks across a set of parallel threads on the GPU (called *device*) (the sequence of work to be done in each thread over a single point in a domain); kernels can be simple functions or complex programs. Threads are grouped into thread blocks with a maximum set of 512 concurrent threads which can efficiently cooperate, communicate, synchronize and share data among themselves through barrier synchronization and a shared access to memory space specific to that block. Each thread within the block has a unique Thread ID (TID) number which can be used to select work and index shared data arrays. In contrast, a grid is a set of thread blocks which can be executed independently and thus in parallel. Thread blocks within a grid are also given unique Block ID numbers. Both grids and thread blocks can be presented in 1-, 2- or 3 dimensional fashion and both represent the domain on which the kernel executes. Since the kernel in its entirety is a sequential code, the actual specification of the dimensions of the grid and thread blocks at kernel invocation time by the programmer explicitly specifies the amount of parallelism. This will allow the programming model to transparently scale to large number of processor cores which could be determined at run time! Furthermore, this will allow determining the number of the thread blocks in a grid based upon the actual amount of data rather than the number of processing cores! Finally, the specification that thread blocks can run independently is what truly allows for scalability! [1][11]

Data communication between threads within a single thread block can be achieved through either the per-block shared memory which is only visible to the threads within the thread block, this memory is usually implemented in SRAM serving a similar function as a first level cache offering low latency (approx. 1 cycle) and fast access, however due to high costs it is often limited to 16kB in recent GPU architectures. Threads can also share data among themselves through the global onboard memory at the price of higher latency (hundreds of cycles)!

Given sufficient hardware resources, independent grids are executed in parallel, but dependent grids are always executed sequentially guarantying that all thread blocks of the first grid are completed before the thread blocks of the next sequential grid are executed through the use of an implicit inter kernel synchronization barrier.

Communication is thus achieved by writing and reading to the global memory. Moreover, within the same grid, thread block communication is not allowed! Threads within a thread block are synchronized entirely by hardware!

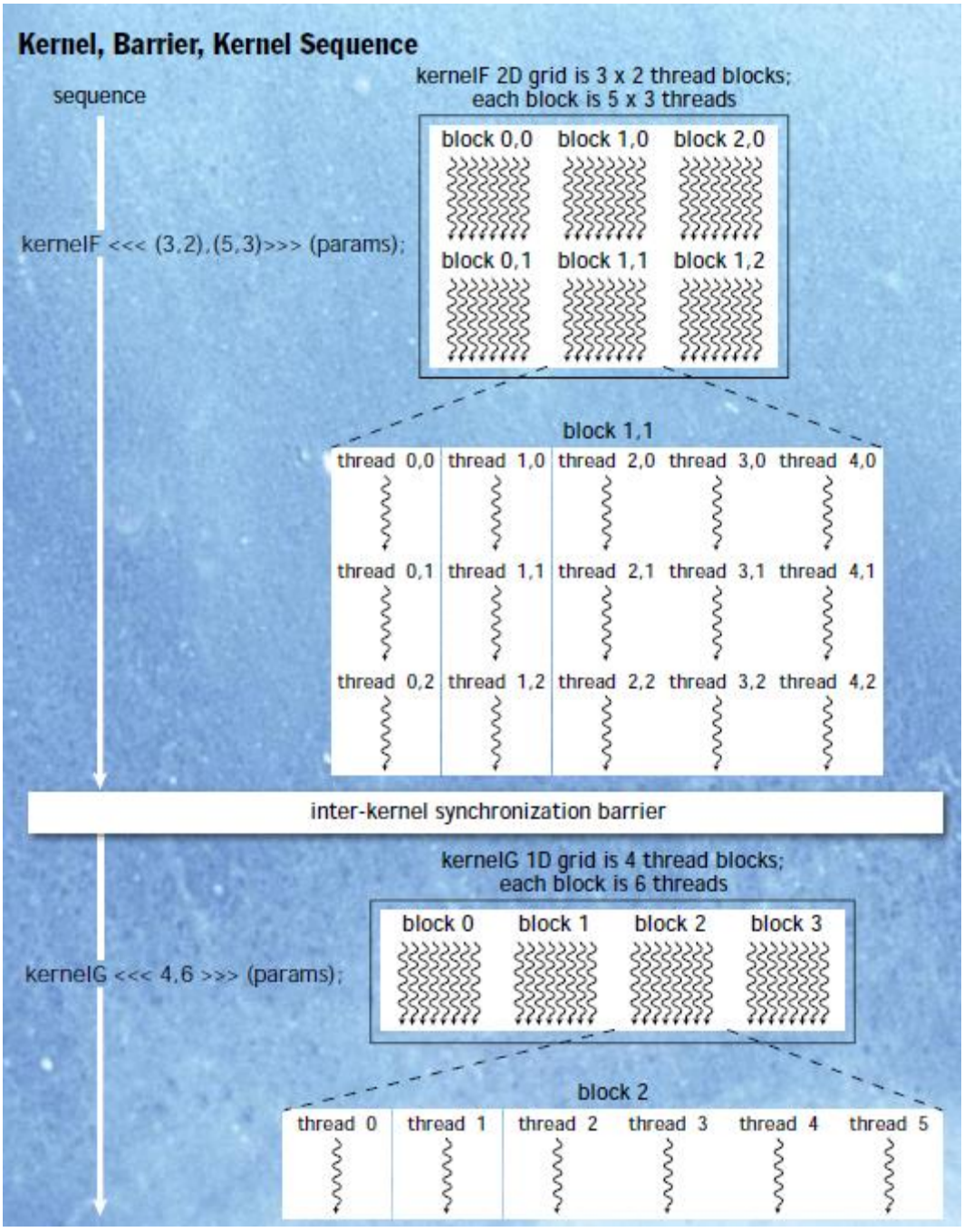

Figure 4. Kernel invocation, sequencing and synchronization barrier in the CUDA programming model

Threads and thread blocks are easily mapped to the hardware architecture discussed in the previous section, initially; the grid configuration is passed to the GPU work distributor through the graphic driver once the kernel is executed. Thread blocks are only assigned to SMs when there are sufficient memory and thread resources where they run as a single unit to completion without preemption. This assignment process is managed by a specialized hardware scheduler. Once assigned, the threads (which could be up to 512) are time sliced onto the 8 SP of the SM in units of 32 threads called warps; each warp is quad-pumped onto the 8 SPs. This multithreading management is all being done in hardware through the use of hardware thread scheduler

embedded in each SM. Figure 4 shows an example of kernel invocation, sequencing and synchronization barrier in the CUDA programming model.

On the level of the thread block, threads express fine-grained data and thread parallelism. Independent thread blocks of a grid express coarse grained data parallelism. Independent grids express coarse grained task parallelism [8]

5. Performance Evaluation

In this section, various performance results and speedups as reported in recent research are presented. Two performance studies summaries are presented; Che et al [1] measured performance and GPU speed up when applications with different communication patterns are involved. Algorithms written in CUDA were compared against a single threaded and four threaded OpenMP implementation both running on the same hardware. Some performance results are highlighted from O. Schenk et al [2] to show the effects of I/O effects in GPU computing performance.

Che et al [1] based their test applications on a subset of domains as presented in the Berkeley's taxonomy (domains are also called dwarves). These test applications mainly differ in their communication patterns, refer to figure 5 below. The *structured grid* dwarf explores computation which is divided into sub-blocks with high spatial locality where the value of every point depends on its neighboring points. Moreover, the connectivity of each point is implicitly defined by its location in the grid. [16] On the other end, unstructured grid dwarves need data points to be explicitly selected based upon the characteristics of the application usually involving multiple memory references indirection for determining the list of neighboring points and loading values. In both preceding dwarves, points in the grid are conceptually updated together. [16] The combinational logic dwarf explores bit-level logic functions and stored state, an example is encryption applications with many bit level permutations. Dynamic programming dwarves solve, store and reuse the results of simple overlapping sub-problems to compute a solution.

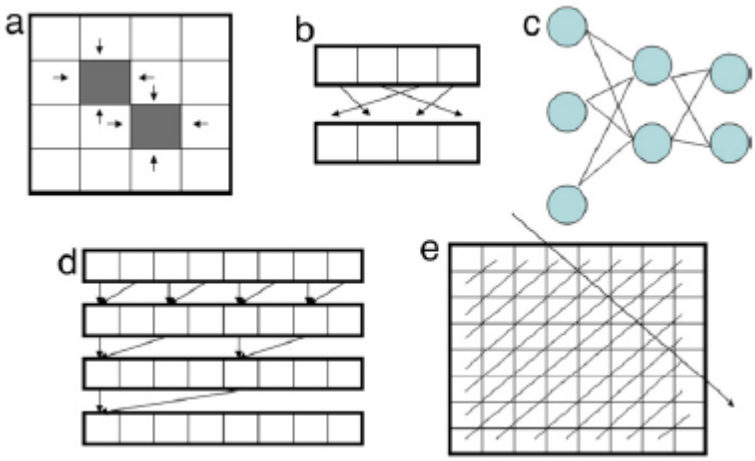

Figure 5. Communication pattern of different applications:
(a) Structured Grid, (b) Bit-level manipulation,
(c) Unstructured Grid, (d) and (e) dynamic programming

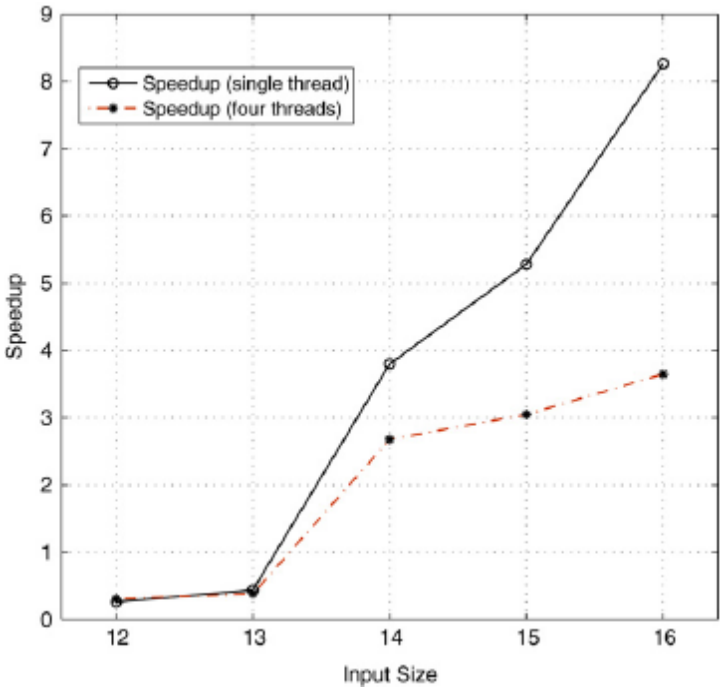

Figure 6. Speed-up of the CUDA version over CPU versions based on a neural network algorithm with unstructured grid communication model. The x-axis is the log2 of the number of inputs

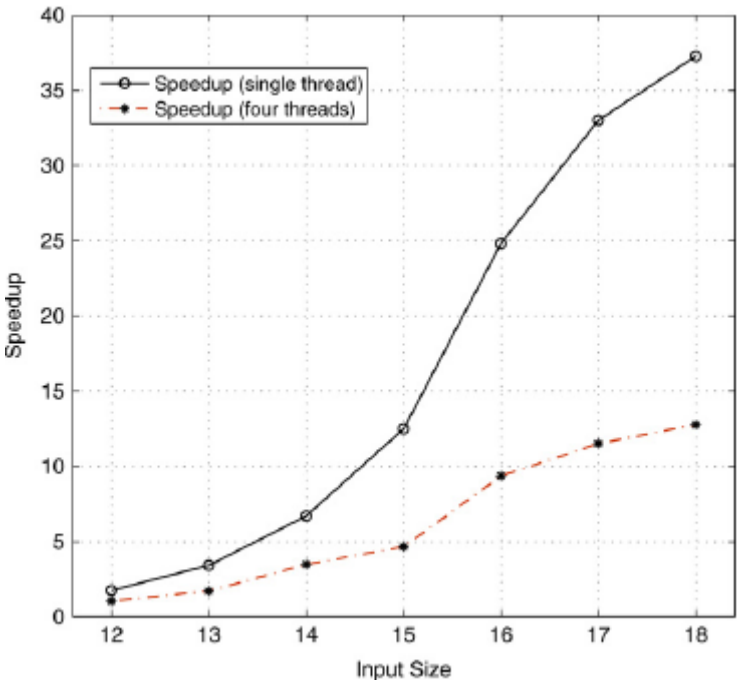

Figure 7. Speed-up of the CUDA version over CPU versions based on a DES algorithm with bit-level communication. The x-axis is the log2 of the number of bits to be encoded

A set of parallel applications encompassing the above mentioned dwarf character tics were run in single threaded and four threaded versions on a dual hyper threaded dual core Intel Xeon processors, and a CUDA version was run on NVIDIA GTX 260 GPU. GPU boasted impressive results. Speedups over single threaded CPU ranged from 2.9X for not inherently parallel algorithm to 37X, meanwhile speedup ranged from 3.5X to 12X over the four-threaded CPU tests. Figures 6 - 8 demonstrate these result graphically

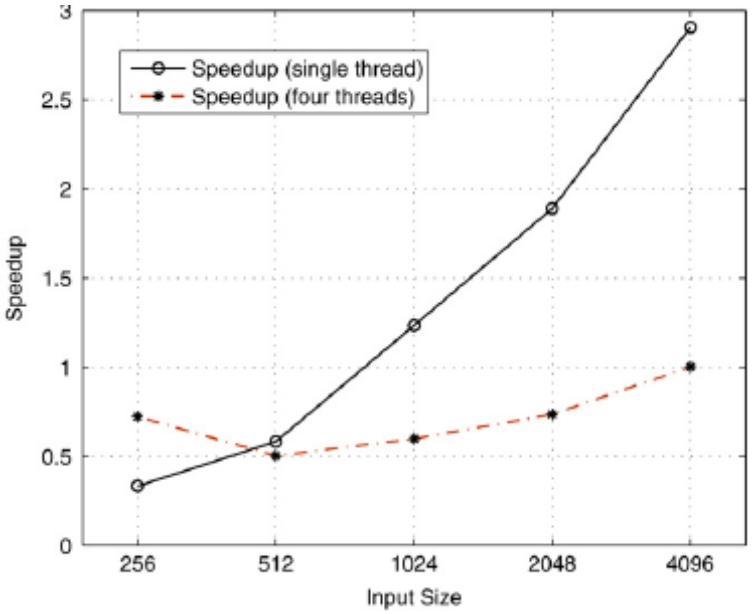

Figure 8. Speed-up of the CUDA version over CPU versions based on an algorithm m exhibiting dynamic programming communication pattern. The x axis represemts the size of the x- and y-dimensions

Schenk et al [2] used basic linear algebra subroutines from Cubla library, a high level API library tuned for performance. Their work primarily dealt with matrix-matrix multiplication, triangular matrix equations, dense and sparse linear factorization solvers subroutines. They compared their results against similar subroutines implementation done with the Intel Math Kernel Libarary (MKL). The CPU versions were run on a 3.4 GHz Intel Pentium D, with 16KB of L1 cache and 2 MB of L2 cache, meanwhile the GPU version was run on NVIDIA GeForce 8800 GTX GPU. All other system specifications are the same. In summary, their results of triangular matrix equations subroutines are discussed to summarize their work.

In Figure 9, the effect of memory transfer overhead (transferring data before computation and results after computation between the main memory and graphics memory) in reducing the performance of GPUs is clear, it is shown that it entails a 10GFLOPs/s reduction in performance. A worthy observation is that CPUs outperform GPUs only when matrices sizes are small even when no memory overhead is involved due to the start up overhead of the many threads to be executed in parallel. Moreover, performance curves show that best results are only achieved when matrices sizes are a multiple of 16 due to GPU memory organization and warp scheduling!

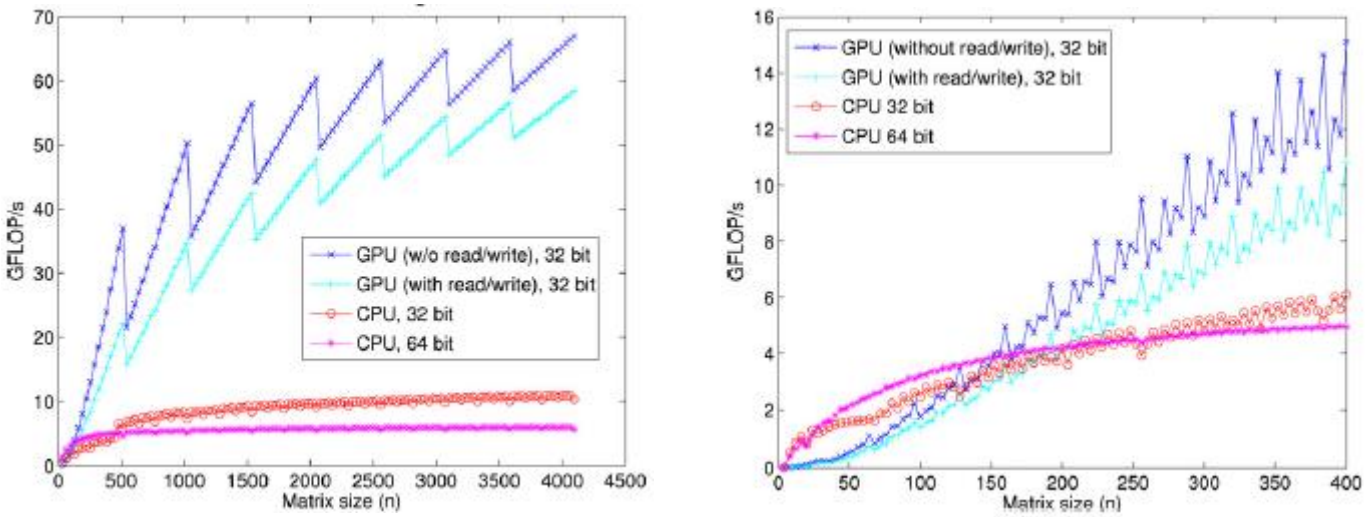

Figure 9. Performance of a triangular matrix equations solver algorithms on CPU and GPU for large matrices n < 4000 (left) and small matrices n<400 (right)

5.1. Hardware Specific Features and Performance Implications

To maximize GPU computing efficiency and achieve higher speed ups and performance gains, programmers need specific knowledge of the underlying hardware architecture and implementation as well as memory hierarchy. Two main hardware associated points severely impact performance if not seriously considered: memory overhead and bank conflicts.

Memory overhead: The total bandwidth in between the on board graphics memory and the GPU is calculated as: *memory partitions x interface width (bytes) x data rate* (i.e. the GeForce 8800 GTX boasts 6 partitions x 8 bytes interface x 1800GHz data rate = 86.4 GB/s total bandwidth)[2]. On the other hand, the graphics card bandwidth to the main memory is restricted by the PCI Express bus through which the card is connected to the system which is rated at 8GB/s for PCI Express version 2.0; that is the data transfer rate between the on-board graphics memory and the GPU is an order of magnitude higher than the peak bandwidth of the

PCI Express bus though it is being done by the fast DMA, and the gap is even widening. In effect, frequent data transfer between the CPU and GPU memories may reduce overall application performance, and since this data communication is explicitly managed by the programmer to minimize bottlenecks, it is advised that data computation and communication be overlapped to reduce memory overhead.

Bank Conflicts: In NVIDIA designs, the 16kB on chip shared memory among threads is organized in 16 banks. If threads access addresses distributed along the same row across all banks simultaneously, no bank conflicts occur and no overhead is incurred. However, if different addresses across different banks are traversed, the accesses will be serialized and maximal bank conflicts will be inflicted! It is the programmers' task to structure the memory accesses such that bank conflicts are avoided or minimized! Figure 10 illustrates that the existence of bank conflicts approximately doubles the kernel's execution time. [1]

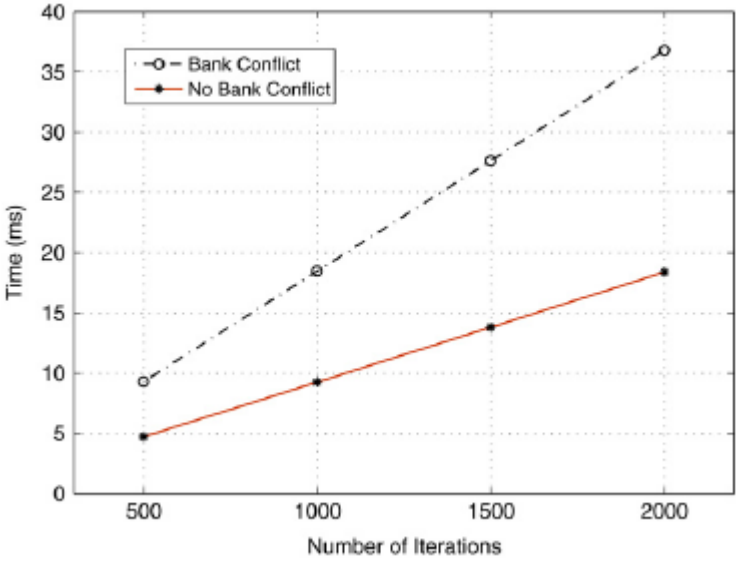

Figure 10. Overhead of bank conflicts

5.2 CUDA Programming Model and Performance Implications

Though the CUDA programming model successfully exploits the inherent application parallelism, abstracts the complexity of the graphical hardware and introduces user friendly high level abstractions and familiar API, those design decisions of the model coupled with the mapping to the underlying hardware affect performance, mainly some overheads are incurred such as control flow, producer – consumer and non contiguous memory access overheads.

Control flow overhead: A drawback of the CUDA programming model is that it is not a purely data-parallel model due to thread divergence issue which causes severe performance penalty and reduced throughput. Thread divergence is caused by control flow instructions (i.e. *if* and *switch* statements) when threads within a warp follow different branches. Divergent branches are executed by serializing each path. Che et al [1] showed that as the number of divergent threads within a warp increases, execution

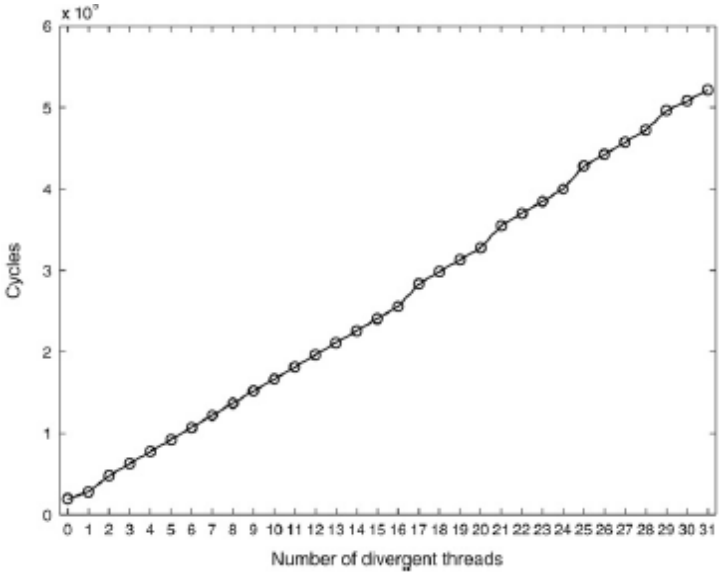

Figure 11. Performance overhead of divergent threads

time increases linearly as shown in Figure 11. To reduce the overhead, programmers should minimize the use of control instructions or ensure that the value of the controlling condition is the same across the entire warp.

Producer- Consumer Overhead: the CUDA programming model specifies that thread blocks are run to completion and leave no persistent state in the per-block shared memory. Therefore, due to this global barrier synchronization in between kernels, in a dependent producer consumer kernel pair relationship, results must be saved and communicated through the global device memory when the dependent consumer kernel is invoked severely hurting performance due to large latency!

Non-contiguous memory access: Since kernels and threads in the CUDA model are scalar, no prior packing is necessitated (packing is grouping of isomorphic scalar instructions that perform the same operation into a vector instruction if dependencies do not prevent doing so [10]). In fact, CUDA allows for non-contiguous memory thread and data accesses even though it reduces the effective memory bandwidth. In this case, packing is regarded as an optimization step.

6. Energy Efficiency

GPUs are often perceived as power hungry devices, in fact state of the art GPUs [i.e. NVIDIA GTX 280] are rated at no minimum of 200 watts power consumption [5] [9]. In [5], Huang et al sought to falsify this notion through an empirical study in which they have shown that contrary to the dominant notion, GPUs are truly energy efficient.

In their work, they compared the energy consumption of a single and multithreaded implementations of a scientific program called GEM targeting a 2.2 GHz Intel Core 2 Duo CPU with 2MB of L2 cache against a CUDA implementation over the state of the art NVIDIA GTX 280 GPU. Execution time and power consumption were profiled for all three implementations and energy efficiency was measured using the energy delay (ED) product.

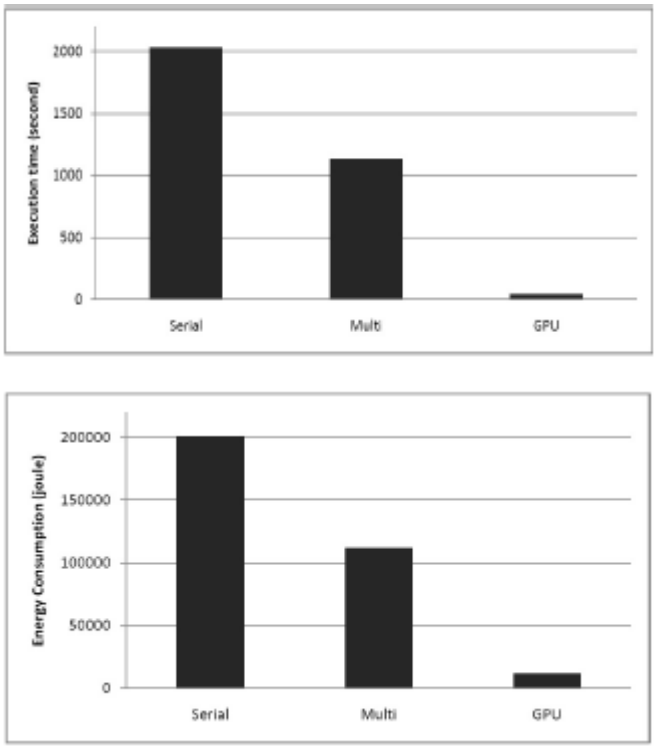

Figure 12. Execution time (right above) and energy consumption (right below) for three implementations of the GEM software as reported in [5]

Figure 12 shows that the GPU implementation of the GEM software is 46 and 25 times faster and offer 17X and 9X better energy consumption than the serial and multithreaded implementations respectively.

Figure 13 plots power consumption (watts) against execution time for the three GEM software implementations. In short, it shows that though the power consumption of the GPU is higher, its execution time is much faster, thus yielding better energy consumption.

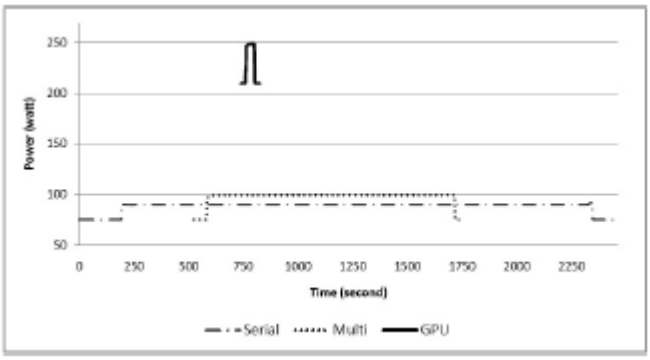

Figure 13 plots the execution times and power consumption of the GEM software [5]

7. The Future and Open Challenges

GPUs are here to stay; they are not to be considered rivals to CPUs but rather as cooperators [8] for each processor has its own application domain at which it excels most. GPUs offer high throughput hiding single thread latency while CPUs excel at sequential applications with high data locality and conditional branches. [3] GPU vendors are introducing more sophisticated designs each year with NVIDIA announcing its new architecture codenamed Fermi, which boasts 512 processing cores, introducing L2 cache for the first time as well as Giga-thread Scheduler. Intel, the leading CPU manufacturer announced its entrance to the GPU domain by introducing the Larrabee GPU, a many core processor based on the x86 instruction set. [7]

Though current programming models for GPU computing are easy to map into hardware and offer high level abstraction to the programmer, they are still imperfect. Data placement management, communication and synchronization prove to be hurdles and increase the complexity in targeting the GPU platform. An open challenge is to develop even higher level programming APIs which promote the use of high level data structures and programming primitives which convey data to the compiler in how to manage concurrency, data placement, communication and synchronization [1]

Moreover, a challenge is to overcome the limitations of the heterogonous memory structure which provides for memory bandwidth bottlenecks and even burden the programmer with writing programs which managing data movement concurrently at program execution.

Furthermore, research should target optimization of hardware organization and thread scheduling in order for performance to scale linearly with thread count and not be restricted to be a multiple of 32 (for best warp scheduling and easy mapping to the 16 banks of PBSM memory).

Little research has been conducted on the optimization of GPU energy performance and efficiency, power and thermal issues will soon prove challenging factors in the design aspects and utilization of GPUs

A final challenge and an interesting area of research is to provide mechanisms to exploit the power of CPUs and GPUs in solving generic problems based on collaborative and a heterogeneous environment [11] with dynamic load balancing based on many criteria such as the size of the problem, scalability on parallel hardware, complexity and even possibly power consumption.

8. Conclusions

The coprocessor trend is returning back. GPUs massively parallel, high processing power, programmability and throughput dramatically reduce time to discovery in high end computation environments. Though, modern graphical processors architectures are built to support GPU computing and are increasingly getting closer in sophistication to modern CPUs, neither GPU vendors nor users are willing to sacrifice the architecture and application domain which made GPUs successful in the first place. [6] Instead, GPU manufacturers are continuously developing general purpose programming models which mask the inherent complexity of the graphical processor. In this report, we walked through GPU hardware organization and design from the early fixed hardwired pipelines to the modern state of the art implementations highlighting the design decisions which made GPUs appealing to general purpose computing in the first place. We also presented two programming models: the inefficient highly restrictive graphical APIs of DirectX and OpenGL, and discussed a successful yet imperfect widely used streaming model implementation provided by NVIDIA: the CUDA programming model which provided good abstraction and easy to use C extensions to facilitate programming.

Since general purpose computing on GPUs trend resurrected due to the high performance gains it offered against CPUs, this report presented, based on the work of [1] and [2], some performance results confirming the superiority of GPUs in high end computing domain. Nevertheless, GPU computing performance is limited by the underlying hardware organization, memory hierarchy and the programming model paradigm. Such issues were also elaborated on in this report.

Furthermore, based on the work of [5], the notion of inefficient and power hungry GPUs was debunked for it has been shown that though GPUs' energy ratings are higher than CPUs, the much reduced execution times for a specific problem in comparison with those run on CPUs argues against the false notion. The report concluded with a list of open challenges and open research domains to further improve GPU computing results.

9. References

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, JW. Sheafer, K. Skadron, *"A Performance Study of General Purpose Applications on Graphics Processors using CUDA ",* Journal of Parallel and Distributed Computing, Volume 68, Issue 10, pp. 1370 – 1380, Elsevier, 2008

[2] O. Schenk, M. Christen, H. Burkhart *"Algorithmic Performance Studies on Graphic Processing Units"* Journal of Parallel and Distributed Computing, Volume 68, Issue 10, pp 1360-1369, Elsevier, 2008

[3] P. Glaskowsky, NVIDIA Whitepaper, "NVIDIA Fermi: The first complete GPU Computing Architecture", 2009

[4] J. D. Owens, D. Luebke, N.Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell. *"A Survey of General-Purpose Computation on Graphics Hardware",* Computer Graphics Forum, Vol 26 No.1 pp. 80–113, March 2007.

[5] S. Huang, S. Xiao, W. Feng , *"On The Energy Efficiency Of Graphics Processing Units For Scientific Computing",* Proceedings of the 2009 IEEE International Symposium, 2009

[6] D. Luebke, G. Humphreys – *"How GPUs Work"* - IEEE Computer, 2007

[7] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan "*Larrabee: A Many-Core X86 Architecture For Visual Computing"* ACM SIGGRAPH, 2008

[8] L. Wang, Y. Huang, X. Chen, C. Zhang – *"Task Scheduling of Parallel Processing in CPU-GPU Collaborative Environment"* Proceedings of the 2008 International Conference on Computer Science and Information Technology, 2008

[9] NVIDIA GTX 280 Page: http://www.nvidia.com/object/product_geforce_gtx_280_us.html

[10] VectorProcessing: http://en.wikipedia.org/wiki/Vectorization_%28computer_science%29

[11] J. Nickolls, I. Buck, M. Garland, K. Skadron ,*"Scalable Parallel Programming with CUDA",* ACM Queue, Vol 6, Issue 2, 2008 , pp 40-53

[12] E. Kilgariff, R. Fernando, *"The GeForce 6 series GPU architecture"* ACM SIGGRAPH. 2005

[13] E Lindholm, J Nickolls, S Oberman, J Montrym , *"NVIDIA Tesla: A Unified Graphics and Computing Architecture"* , IEEE Micro, 2008

[14] NVIDIA Whitepaper, *"NVIDIA's Next Generation CUDA Compute Architecture: Fermi",* 2009

[15] Dana Schaa, David Kaeli, "Exploring the multiple-GPU design space," ipdps, pp.1-12, 2009 IEEE International Symposium on Parallel&Distributed Processing, 2009

[16] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A.Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, *"The landscape of parallel computing research: A view from Berkeley, Tech. Rep."* UCB/EECS- 2006-183, Department of Electrical Engineering and Computer Sciences,University of California, Berkeley, Dec 2006.

[17] NVIDIA Technical Brief, *"NVIDIA GEFORCE GTX 200 GPU Architectural Overview",* 2008