Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

A Review on GPU Programming Strategies and Recent Trends in GPU Computing

Nisha Chandran S.*, Durgaprasad Gangodkar, Ankush Mittal

Department of Computer Science and Engineering
Graphic Era Deemed to be University, Dehradun, Uttarakhand, India
*Corresponding author: nisha.unnikrishnan@gmail.com

(Received December 20, 2017; Accepted August 6, 2018)

Abstract
The advancements in the field of internet and cloud computing has resulted in a huge amount of multimedia data and processing of this data have become more complex and computationally intensive. With the advent of the scalable, inexpensive Graphics Processing Units (GPUs) with very high computation power, the processing of such big data has become less expensive and efficient. Also fast developments happening in the field of programming languages and different programming and debugging tools adds to the ease of GPU programming. However, utilizing the resources of the GPU effectively and fully is still a challenge. The goal of this paper is to present a brief review of NVIDIA's state of the art Fermi architecture and to survey different programming and optimization strategies adopted by researchers' to accelerate the GPU computation. This survey aims to provide researchers with knowledge about the different programming and optimization techniques in GPU programming and to motivate them to architect highly efficient parallel algorithms by extracting maximum available capability of the GPUs. The paper also explores some recent trends in the field of GPU programming.

Keywords- Parallel Programming, GPUs, CUDA, Fermi, Debugging, Programming Strategies, Optimization.

1. Introduction

Graphics Processing Unit (GPU) has entered the General Purpose Computing Domain (GPGPU) for over a decade now. In comparison to the single processor CPU, GPGPUs have very high computation power. According to the literature (Brodtkorb et al., 2013; Kirk et al., 2016) when comparing the theoretical peak bandwidth and the gigaflops performance there is a huge gap between CPU and the GPU. CPUs basically are optimized for executing a series of operations in order. Modern CPUs though are very efficient in terms of flexibility and performance; they have very complex control hardware and are very expensive in terms of power. GPUs on the other hand have very simple control hardware and are more power efficient. While the CPUs are optimized for the latency, the GPUs concentrate on optimizing the throughput. Even though other types of accelerator cores like Field Programmable Gate Arrays (FPGAs) and Cell Broadband Engines (Cell BEs) are available, GPUs have gained much popularity than these in the last few decades. This is because programming FPGAs and Cell BEs for general purpose computing is much difficult than GPUs and compared to them GPUs are easily available as most of the recent desktop and laptop computers now come with a dedicated GPU (Brodtkorb, et al., 2013). Currently, there are three major vendors of GPU

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

in PC market and they are Intel, AMD and NVIDIA. However, NVIDIA is the most dominant vendor in the academic environment and therefore we will be focussing on NVIDIA GPUs and NVIDIA CUDA as the programming language in this paper.

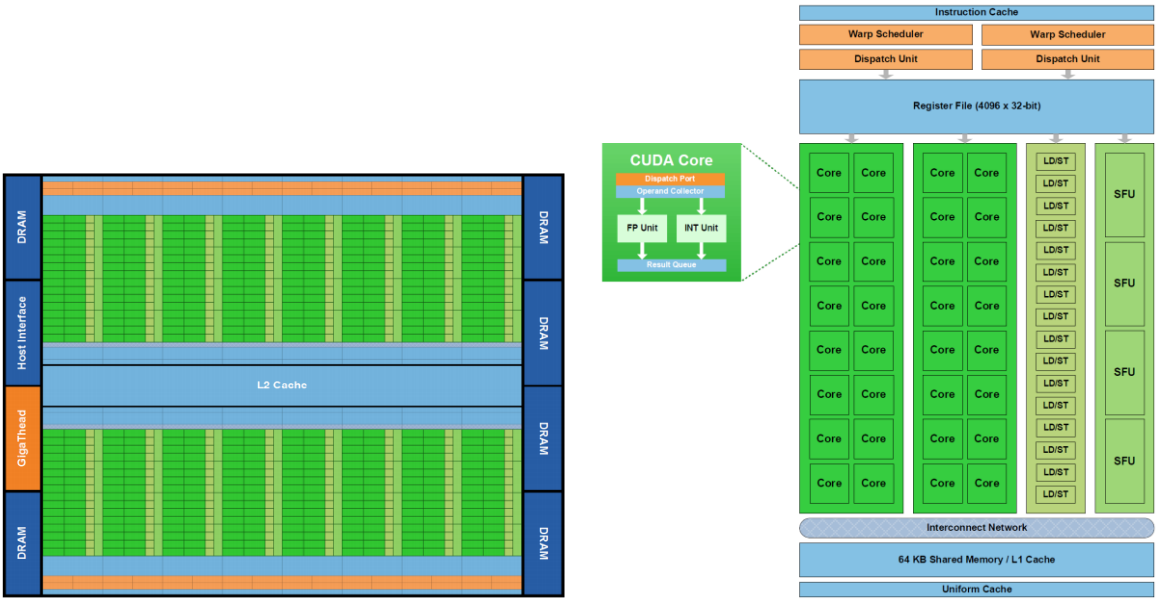
Traditional GPGPU development was based on graphics function library like OpenGL and Direct 3D and was largely used by professionals, who were familiar with the graphics API. However, the emergence of Compute Unified Device Architecture (CUDA), NVIDIA's parallel architecture implementation, removed these inconveniences as it provides APIs for programmers to develop parallel applications using the C programming language. The programmers write C programs with CUDA extensions and target a general purpose massively parallel processor (NVIDIA 2010) and thus a dramatic increase in computing performance is achieved by harnessing the power of the GPU. The aim of this paper is to give the readers a brief overview of the GPU programming model, debugging tools, programming as well as optimization techniques adopted for the GPU codes. First we start with a short overview of the state of the art Femi GPU hardware in section 2. Further, in section 3 we give an overview of the programming model and the commonly used debugging tools. A detail review of the common programming strategies adopted by the GPU programmers is given in section 4. In section 5 we present a detail review of the different optimization strategies usually adopted in the GPU programming. In section 6 we discuss some of the recent trends in GPU computing and different GPU architectures launched after the Fermi architecture. Finally, in section 7 we conclude the paper.

2. Fermi GPU Architecture

Fermi architecture is considered as NVIDIA's first complete GPU (Patterson, 2009; Wittenbrink et al., 2011) as it delivered almost all of the features required for the most demanding high performance computing applications. It was the most significant leap forward since the G80 architecture. Figure 1a shows the high level block diagram of the first Fermi chip. As shown in figure the Fermi architecture consists of 512 accelerator cores called CUDA cores. Each core includes a fully pipelined integer arithmetic and logic unit and floating point unit that execute one integer or floating point operation per clock cycle. Each CUDA core is organized into 16 Streaming Multiprocessors (SMs) each with 32 CUDA cores. There is 768KB L2 cache shared across all 16 multiprocessors and a 384-bit GDDR5 DRAM memory interface. The host interface shown in Figure 1a is used to connect the GPU to the CPU via PCI-express bus. Further the Giga Thread global scheduler distributes the thread blocks to the multiprocessor thread schedulers. Figure 1b shows a single SM consisting of 32 cores where each of them can execute one floating point or integer instruction per clock. Each SM also has 16 load-store units for memory operations, four special-function units, a 4K word register file, and 64K of local SRAM split between cache and local memory. Special Function Units (SFUs) are used to execute instructions like sine, cosine, square root and interpolation. The threads are scheduled in groups of 32 parallel threads called warps. There are two warp schedulers and two instruction dispatch units as shown in Figure 1b. This allows two warps to be issued and executed concurrently. Further,

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

there is a 64KB on chip memory which can be configured as 48KB of shared memory and 16KB of L1 cache or vice versa (Patterson, 2009; Wittenbrink et al., 2011; Brodtkorb et al., 2013).



(a) **(b)**

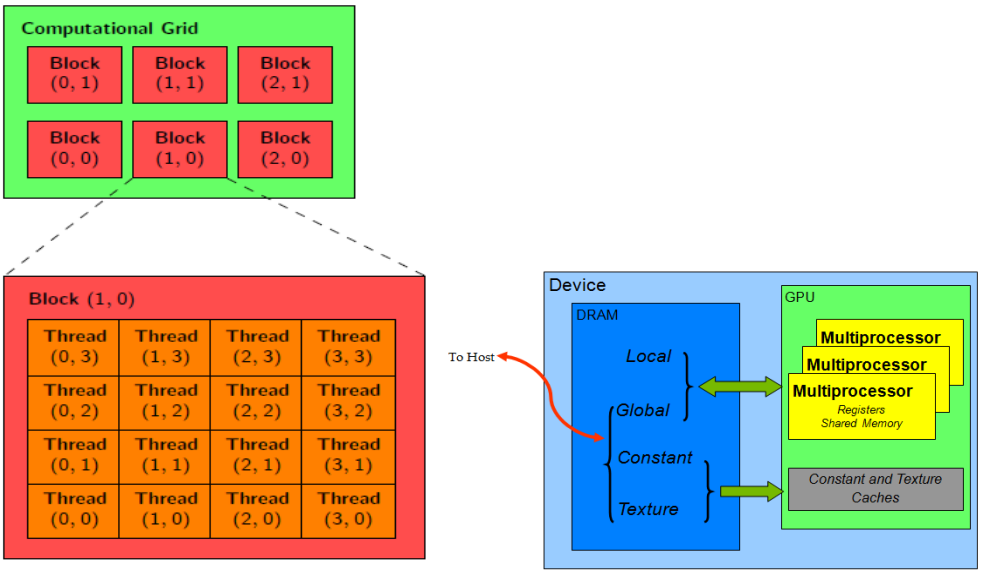
Figure 1. a) Fermi Architecture consisting of 16 Streaming Multiprocessors (SMs) b) Single SM (Patterson, 2009; Wittenbrink et al., 2011)

3. GPU Programming Model and Debugging Tools

CUDA programs are written in 'C for CUDA', which is a subset of C with extensions for executing functions in parallel. The programs are compiled using nvcc, NVIDIA's CUDA compiler. A CUDA program calls parallel kernels which executes, in parallel, across a set of parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks. A kernel program is instantiated on the GPU on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results. A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. Every thread block has a unique block ID within its grid. A grid is an array of thread blocks that execute the same kernel, reads inputs from global memory, writes results to global memory, and synchronizes between different kernel calls. There is a per-thread private memory space used for register spills, function calls and C automatic array variables for every thread in the CUDA parallel programming model. Each thread block has a per-block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in global memory space after kernel-wide global

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

synchronization. CUDA concept of grid of blocks is as shown in Figure 2a. The figure shows the 2D hierarchy of blocks and threads normally used to process an image. Programmer defines the required number of thread blocks and it is the GPU which decides which thread blocks to be run on which SMs. This abstraction is one of the biggest advantages of CUDA as the hardware can run things independently and efficiently. CUDA guarantees that all threads in a block run on the same SM at the same time and that all blocks of a kernel finish execution before executing the next kernel.



(a) **(b)**

Figure 2a. CUDA concept of a grid of blocks (Kirk and Wen-Mei, 2016; Wen-Mei, 2011), 2.b. Device Memory Spaces (Wen-Mei, 2011)

Figure 2b shows the CUDA memory model. CUDA devices use several memory spaces, which have different characteristics that reflect their distinct usages in CUDA applications. These memory spaces include global, local, shared, texture, and registers. Of these different memory spaces, global memory is the most plentiful. Global, local, and texture memory have the greatest access latency, followed by constant memory, shared memory, and the register file. The various principal traits of the memory types are shown in Table 1.

Table 1. Memory types in NVIDIA CUDA and their principal traits (Wen-Mei, 2011)

Memory	Location on/off chip	Access	Scope	Lifetime
Register	On	R/W	1 thread	Thread
Local	Off	R/W	1 thread	Thread
Shared	On	R/W	All threads in block	Block
Global	Off	R/W	All threads and host	Host allocation
Constant	Off	R	All threads and host	Host allocation
Texture	Off	R	All threads and host	Host allocation

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

In this section the debugging tools provided by NVIDIA to support the CUDA programming is reviewed. There are several powerful CUDA debugging tools that support most of the commonly used operating systems. Some examples of such debugging tools are Parallel NSight for Microsoft windows and CUDA-GDB available for Linux and Mac. CUDA-GDB provides developers a mechanism for debugging a CUDA application on actual hardware at real-time. Some features of the CUDA-GDB are listed below (CUDA-GDB NVIDIA CUDA Debugger, 2010)

- user can inspect a specific host thread or a CUDA thread and switch focus to either
- allows the execution to be paused at any function symbol or source file line number
- supports stepping GPU code at the finest granularity of a warp
- address of any variable can be printed to find out its storage location whether it is local, shared or global.
- inspection of any kernel possible
- supports breaking into hanging or indefinitely looping kernels
- supports in checking memory error using the command *set cuda memcheck on.*

Parallel NSight provides conditional breakpoints, assembly level debugging, and memory checking and also it is freely available for use. Some features available on Parallel NSight are listed below (S3478-Debugging CUDA kernel code, 2013).

- supports inspecting variable values
- supports inspecting memory values
- supports setting both source and data breakpoints
- supports specifying the debugger context as block level or thread level
- helps to setup local headless GPU debugging in case of two GPUs
- massively threaded kernel navigation and run control
- CUDA memory checker
- CUDA information tool window showing all CUDA resources.

4. Review on GPU Programming Strategies

Programming with GPU and getting a speedup much better than many existing CPU code is a relatively easy task but extracting maximum advantage of the hardware is really a challenge. Different programming strategies are adopted by programmers to utilize the hardware available to the maximum. In this section we review some of the programming strategies adopted by researchers in order to accelerate the CUDA application (Cornel virtual workshop, 2013; Pan American Advanced Study Institute, 2011).

4.1 Latency Hiding

GPUs issues instruction in order and when the issue stalls for arguments the GPUs switch between threads to hide the math latency. This strategy of the GPU to instantly switch between threads can be done efficiently only if there are enough threads launched for the execution. This measure of the maximum number of threads that a GPU can run concurrently is referred to as occupancy. Several works have been done by researchers to improve the

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

latency hiding in GPUs. In (Lee and Wu, 2014) a latency profiling approach is used for effective evaluation of the latency hiding characteristics of the Fermi architecture. From their study the authors found that for certain GPU applications like Breadth First Search (BFS) even though the context switching effectively hides the latency the performance is limited by other factors. In (Kim et al., 2016) a different approach which makes uses of the warp pre-execution mode is adopted to improve the latency hiding. In the method adopted instructions which are independent of the long latency operations are pre-executed while the long latency dependent operations are skipped. However, the authors found it very challenging to maintain the overall sequential semantics of the program. Although by increasing occupancy the memory latencies can be hidden, higher occupancy does not always give high performance. Once the memory latencies are hidden there is little performance gain by increasing the occupancy. The rule of thumb is that there should be a minimum of 1000 threads per GPU or approximately 512 threads per SM.

4.2 Thread Divergence

Another issue which affects GPU performance is the thread divergence. GPUs execute the instructions in a 32 way Single Instruction Multiple Data (SIMD) fashion in which a single instruction is issued for a warp (thread vector) of 32 threads. The threads within a warp must execute the same instruction at the same time or in other words thread divergence must be avoided to ensure good performance. The common scenario of a control flow statement 'if-then-else' in the program can cause thread divergence. Threads in a warp will be required to diverge to evaluate the true and the false condition of the if-then-else. Because of the restriction that the threads in a warp cannot diverge, the threads executing the true and the false condition will not be executing in parallel as expected. While the threads executing the 'then' are active the remaining threads in the warp which evaluated to 'else' condition will be deactivated and vice versa. This serialization of the threads will result in performance degradation. Thread divergence can also result in a deadlock in some cases where there are __syncthread() barriers in an 'if-then-else' statement as shown in the example program below (Cornel virtual workshop, 2013; Pan American Advanced Study Institute, 2011)

```
if  (threadIdx.x<256){program_then();__syncthreads();}else if (threadIdx.x >=256)
                                {program_else();__syncthreads();}
```

Therefore, to ensure efficiently performing kernels expensive branching operations must be avoided and the threads in warp must be optimized to perform the same operation without much branching. Some notable techniques adopted by researchers to avoid thread divergence is reviewed here. In (Zhang et al., 2010) a run time thread data remapping scheme is adopted to handle the thread divergence efficiently. The data sets on which the threads in a warp work are switched and the whole warp of threads are made to take the same execution path on a conditional branch. Another work in which the data reordering scheme is used for addressing the thread divergence problem is given in (Chakroun et al., 2013) in which they parallelize

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

the branch and bound algorithm for solving Flow-shop Scheduling optimization Problems (FSP).

4.3 Memory Coalescing

GPUs perform most efficiently when threads operate on contiguous memory locations i.e. when the memory accesses are coalesced. Memory accesses become serialized in case of misaligned accesses, sparse memory accesses or non sequential memory accesses and this will affect the performance largely. The example code below shows both coalesced and non coalesced memory operations (Cornel virtual workshop, 2013; Pan American Advanced Study Institute, 2011).

```
__global__ void program_mem(float *g){ float a=3.14;int i= threadIdx.x; g[i]=a;
                                        g [i*2]=a;}
```

The most common method adapted to misaligned memory accesses is data reorganisation. In (Wu et al., 2013) two data reorganisation algorithms are proposed padding and sharing algorithm. In padding algorithm the memory segments are padded with empty slots thereby making the accesses to the segment coalesced. However, this method since reorganizes the threads along with the data it can affect other references in the kernel. The sharing algorithm operates by shifting all the non-coalesced access from global memory to shared memory thereby reducing the duplication of the data. In (Fauzia et al., 2015) a dynamic tool for analyzing uncoalesced memory accesses and a framework which remaps the work among the threads in a warp is proposed to avoid the uncoalesced memory accesses.

4.4 Data Reuse

Another factor which affects the GPU performance is the transferring of the data from the memory to the processor. Most of the algorithms have heavy memory operations compared to the computation operations. The only option available in such cases is to reuse the data and keep the frequently used data in the fastest memory available in the device. GPUs have three fast memories available which are local, shared and global memory. Local memory which is the registers and the L1 cache is the fastest memory and is private to a thread. GPUs have limited set of registers available and so allocating large number of threads with the intention of increasing occupancy will actually affect the performance as there aren't enough registers to cater to these threads. Next fastest memory is the shared memory which is shared by all the threads in a block. With proper usage shared memory can function as fast as registers, however, shared memory is also limited like registers to normally (48KB in Fermi) per SM. To make proper use of the shared memory data is divided into tiles and the tiles are loaded into the memory and operated upon from there. Further, shared memory is divided into equally sized smaller sub arrays called banks and these banks can be accessed simultaneously. This provides high memory bandwidth as any memory load or store of specific number of addresses that span the same number of banks can be serviced

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

concurrently. If multiple threads' requested addresses map to the same memory bank a bank conflict arises thereby making the accesses serial. Different memory access patterns which results and do not results in bank conflicts is shown in Figure 3. As shown in Figure 3a left and right figures linear addressing with a stride of one 32 bit word and three 32 bit word respectively will not result in a bank conflict, whereas in the middle figure linear addressing with a stride of two 32 bit word is used and therefore results in a two way bank conflict. However, using random permutations and broadcasting, bank conflicts can be avoided as shown in Figure 3b. Bank conflict can also be avoided by the technique of memory padding. For example, the codes below show a shared memory declaration which can result in bank conflict and the modified code which avoids the conflict by padding with one element.

```
__shared__int sh_mem[tile_width][tile_height] // can result in bank conflict
__shared__int sh_mem[tile_width][tile_height+1] //padded to avoid bank conflict
```

Another important point to remember when using shared memory is to avoid the race conditions between threads. Thread barriers like __syncthreads () and atomic operations are normally used to avoid the race conditions and to ensure that when one thread is accessing one memory location no other operation is occurring in the same location. However, atomic operations are slower as they serialize the execution.

4.5. Streams

It is also possible to enhance parallelism in CUDA by launching multiple kernels in parallel using CUDA streams. By increasing the number of concurrent streams a higher degree of parallelism can be achieved. Stream is an in-order queue of operations which includes kernel launches and memory transfers that will be executed by the GPU. Figure 4 shows n+1 independent streams running in parallel Streams are useful in doing heterogeneous computing in which the CPU and GPU works concurrently. While the GPU is busy executing kernel launches and memory transfers, the CPU continues to perform its own operations and on completion, synchronizes with the GPU for the results. Thus along with the data parallelism which is achieved using threads and blocks task parallelism can also be achieved in GPU programming using streams.

5. Review on GPU Optimization Strategies

First step to be followed when optimizing a CUDA program is to identify the performance bottlenecks within the program. Three major optimizations considered by default for a GPU program is the kernel optimization, memory optimization and latency optimization (Cornel virtual workshop, 2013; Pan American Advanced Study Institute, 2011; Patterson, 2009; Wittenbrink et al., 2011; CUDA Optimization Techniques 2010).

5.1 Kernel Optimization

The CUDA visual profiler tool can be used for identifying the bottlenecks in a CUDA kernel. First step is to check using the visual profiler whether the kernel is bandwidth bound or

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

compute bound. For bandwidth bound kernels some optimizations to be considered are avoiding global memory coalescing, usage of shared memory as programmer designed cache whenever possible, considering using structure of arrays data structures. For compute bound kernels which are less likely, some strength reduction of the instructions can be done like replacing a multiplication operation by shift or addition operation or by replacing a division operation by reciprocal multiplication. Reduction in the overall number of operations might also help. Expensive re-computation can be reduced by pre-computing and storing values in temporary variables. Further, kernels must be made as coarse-grained as possible so that maximum amount of work can be done in limited kernel calls and the overhead of launching a kernel can be reduced. Limiting the shared memory and register usage within the kernel also can be tried thereby increasing the occupancy of the kernels. In (Lee et al., 2012) some efficient kernel optimization strategies are proposed for neuroimaging algorithms. They have optimized the compute bound kernels by reducing the use of registers and by increasing the data throughput by increasing workload of the threads. For memory bound kernels data is reorganized into self contained structures and a multi pass approach is adopted for efficient optimization. Kernel launch configuration can also be optimized by adjusting the number of blocks and the number of threads within each block in such a way that the device is utilized to the maximum. There must be enough independent threads to hide the instruction and memory latencies. Avoiding thread divergence is also an important technique used to optimize the kernel.

5.2 Memory Optimization

Two major factors to be considered while optimizing memory is memory access patterns and the number of concurrent memory requests. Unlike CPUs which are designed in such a way that slightly irregular memory access patterns do not affect the performance, in GPUs the same access patterns might affect the performance badly. By taking advantage of the GPU's specialized address spaces like constant and texture memory which is based on spatial and temporal locality, this limitation can be addressed. For very frequently used data, using shared memory which is slightly limited in size can also be considered. Memory coalescing is another technique to be considered when global memory is used. However, it is advantages to minimize the usage of global memory and maximize the usage of shared memory without bank conflicts wherever possible. Another important point to be considered while optimizing memory is to reduce the overhead of memory transfers between the host and the device. Programmers must keep in mind to perform maximum computations in the device so that the frequent memory transfers between the CPU and the GPU can be reduced. Programmers should also keep in mind that the memory management operation in CUDA which are *cudaMalloc* and *cudaFree* are expensive operations compared to their C counter parts *malloc* and *free*. Therefore, to reduce the usage of these operations it is always advantageous to allocate memory once in the starting of the operation and keep on reusing the memory for every kernel calls. Streams, which are a sequence of operations that execute in the issuing

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

order, can be used to overlap the memory operations and the kernel launches for providing additional asynchronicity thereby improving performance.

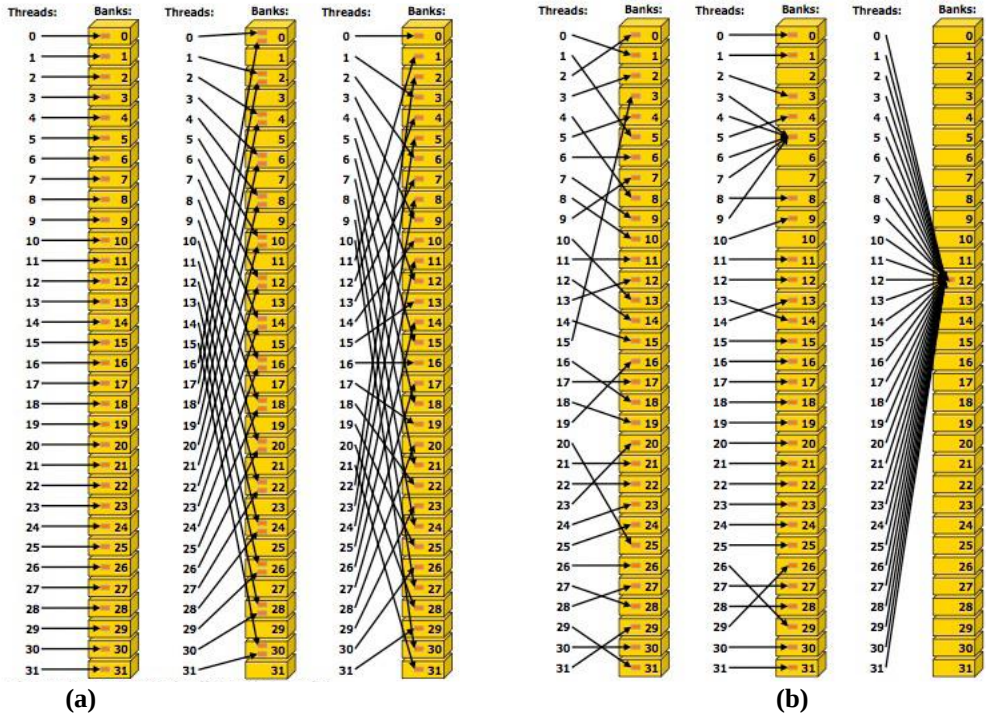


Figure 3.a. Left: linear addressing with stride of one 32 bit word (no bank conflict), Middle : linear addressing with stride of two 32 bit word (2 way bank conflict), Right: linear addressing with stride of three 32 bit word (no bank conflict), 3.b. Left: conflict free access via random permutations, Middle: conflict free access since threads 3,4,6,7 and 9 access the same word within bank 5, Right: conflict free broadcast access (all threads access the same word) (Wen-Mei, 2011)

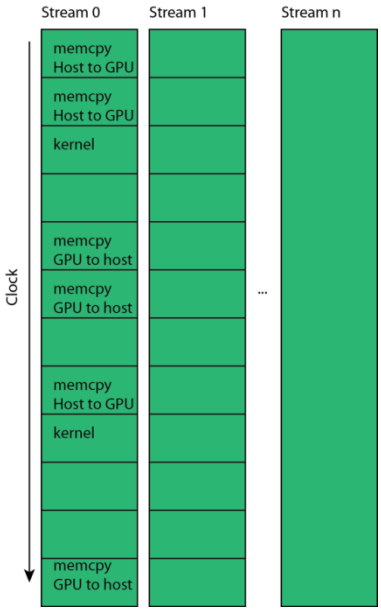


Figure 4. CUDA streams (Wen-Mei, 2011)

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

In (Li et al., 2016) memory efficiency issue of deep Convolutional Neural Network (CNN) implementations in GPU is studied. The memory access patterns of the different memory bound CNN layers are analysed and are efficiently optimized to reduce the off chip memory access and the inter kernel communication. Authors in (Siegel et al., 2011) proposed a memory layout optimization method in their parallel version of the real world Gravit application. They split the large memory structures into smaller sub structures and aligned them consecutively in the global memory. A performance improvement of 50% was achieved compared to the unoptimized layout of the application.

5.3 Optimizing Latency and Instruction Throughput

Warp serialization or shared memory bank conflicts can be the main reason for an instruction throughput bounded kernel. The CUDA visual profiler can be used to determine these problems. Determining the difference between the *instructions_executed* counter and the *instructions_issued* counter gives an idea about the serialization happening in the kernel. Thread divergence can be determined by comparing the *divergent_branch* counter to the *branch* counter and if there is a divergence the source code can be modified to eliminate the divergence (Brodtkorb et al., 2013). Shared memory bank conflicts can be eliminated by the padding technique. Also source code can be modified in such a way to make all the memory accesses broadcasts or bank conflict free. To determine whether the kernel is bounded by arithmetic operation latencies or data dependencies we will have to compare the kernel performance with the hardware limits using the IPC-Instructions/cycle counter. A low value of this shows there are data dependencies and arithmetic latencies (Brodtkorb et at., 2013). Authors in Ref. (Tan et al., 2011) fine tune the double precision matrix multiplication algorithm using instruction scheduling derived entirely from instruction dependency. They could successfully reduce the memory latency and increase the instruction throughput to achieve a major performance improvement.

6. Recent Trends in GPU Computing

Major success of GPUs in general purpose computing owes to the fact that they are very inexpensive. The two major vendors of the GPUs NVIDIA and AMD develops GPUs on a large scale for the entertainment industry and with the advent of using GPUs for general purpose computing they have added additional functionality to the GPUs to cater to this new trend. Thus GPUs are now available in almost everything from cell phones to supercomputers. GPU based parallelization is now being applied in almost all fields of scientific computing ranging from image and video processing, remote sensing, machine learning, operations research, data mining etc. We review some recent research works related to GPU computing spanning varied areas of research. A Google scholar search on GPU computing resulted in a lot of papers dedicated to GPU computing in different areas of scientific research. Large amount of work from remote sensing, medicine, molecular biology, and artificial intelligence fields are being done on GPUs recently. In (Ma et al., 2016**)** a reusable GPU based parallel processing image model was proposed for remote sensing applications. In (Ke, et al., 2016) a parallel computing framework for cloud filtering and

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

smoothing for remote sensing images was proposed. Further, a Kepler compute architecture based GPU was used for detecting oil spill detection from multi-temporal LANDSAT-7 imagery (Bhangale et al., 2017). Most of these algorithms have attained significant speedup compared to their CPU counterpart. GPU programming is widely being used in medicine and molecular biology fields. Single-particle cryo-EM structure determination is a new trend which is transforming structural biology. GPUs are being used to attain significant speedup in the image classification and high resolution refinement steps involved in the cryo-EM structure determination workflow (Kimanius et al., 2016). Multiple sequence alignment is a highly intensive computational problem in computational molecular biology where a similar DNA sequences are aligned and a molecular function prediction is done. In (Chen et al., 2017) a CPU/GPU heterogeneous platform is used to build such an alignment system. In (Sundfeld et al., 2017) the very first GPU based solution for RNA structural alignment problem based on Sankoff algorithm is proposed. In (Dubey et al., 2016) GPUs are being used for ab initio protein structure prediction problem, which is a computational protein structure prediction from its primary amino acid sequence. It is a computationally very expensive algorithm and the authors have gained significant gain in computational time when done on GPUs. Research works related to other fields are also currently done on GPUs to speed up the intense computations. One such work in the area of Physics is Feynman Integral Evaluation by a Sector Decomposition Approach (FIESTA) (Smirnov, 2016) a new algorithm for better optical performance in integral evaluation. It aims at a calculation with increased number of sampling points to reduce the uncertainty estimates. In (Mantas et al., 2016) a GPU based implementation of several simple numerical examples of partial differential equations is studied to give an idea of how efficiently such computationally intensive mathematical problems can be done on a GPU platform. (Jung and Bae, 2018) proposes the GPU implementation of a new direct linear equation solver that can be applied to mechanical system analysis. (Domínguez et al., 2016) proposes the GPU implementation of the computationally expensive Smoothed Particle Hydrodynamics (SPH), a numerical method suitable for describing a variety of complex free-surface flows with large discontinuities. A parallel implementation of the most widely used machine learning algorithms Elastic Net and Lasso algorithms is presented in (Zhou et al., 2015). (Wu et al., 2017) proposes a model named VLogGP to study the behaviour of parallel applications specifically, the communication and memory access patterns for CPU/GPU heterogeneous systems. In (Doulgerakis et al., 2017) a GPU implementation of the highly computational parameter recovery in diffuse optical tomography is presented. The method is said to be highly applicable for both continuous wave and frequency domain systems and has achieved almost 10 times speed increase when done on GPUs. Currently, NVIDIA GPUs are also on the forefront in accelerating many deep neural networks and artificial intelligence applications by a factor of 10 to 20x compared to the CPU, reducing the training time from weeks to days.

Next we review the GPU architectures developed by NVIDIA after the Fermi architecture and the new functionalities added to each of them to enhance the efficiency of the general

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

purpose GPU computing. The immediate successor of Fermi, the Kepler architecture (NVIDIA Kepler GK110, 2013) has undergone a major change in the streaming multiprocessor organization (now called SMX) with just four multiprocessors and each with 192 CUDA cores (1536 CUDA cores on one chip). Also the clock frequency was decreased from 1.5GHz to 1 GHz. This was all aimed at achieving increasing performance through more cores running at a decreased clock frequency. Compared to Fermi the bandwidth of the L2 cache was also increased to 512KB to cater to applications that uses a large amount of L2 cache. Each SMX in Kepler features four warp schedulers and eight instruction dispatch units, allowing four warps of 32 parallel threads to be issued and executed concurrently. The number of registers per thread was quadrupled to 255. Memory configuration is similar to Fermi with an additional split for 64KB shared memory to 32KB each between shared memory and L1 cache. A new feature called dynamic parallelism that allows the GPU to generate new work for itself without the involvement of CPU was also introduced in Kepler. Maxwell architecture (NVIDIA Maxwell GM204 Architecture, 2016) the successor of Kepler provided a big leap in power efficiency and performance compared to the previous generations. It also delivers 2x the performance per watt compared to the Kepler products. Maxwell architecture consists of 16SMs (now called SMMs) with 128 CUDA cores (2048 CUDA cores on one chip) and 128 texture units. Memory bandwidth was increased from Kepler's 192GB/sec to 224GB/sec and L2 cache size was increased to 2048KB. Each Maxwell SMM contains four warp schedulers capable of dispatching two instructions per clock cycle. Compared to Kepler the memory hierarchy has also changed by implementing 96Kb dedicated shared memory, while the L1 cache is combined with the texture cache function. Each Maxwell CUDA core with a larger dedicated shared memory and a larger cache is able to deliver roughly 1.4x more performance per core compared to Kepler. In addition to the power and computational performance improvement, Maxwell also provides some other features like NVIDIA Voxel Global Illumination (VXGI), Multi Frame sampled Anti- Aliasing (MFAA), dynamic super resolution, conservative rasterization, viewport multicast and sparse texture. Tesla P100 (NVIDIA Tesla P100 whitepaper, 2016) the successor to Maxwell with the Pascal architecture is featured as the world's fastest GPU with 15.3 billion transistors. The most important feature of this GPU is the new high speed interface NVLink that provides GPU to GPU data transfer at up to 160GB/sec. It provides single, seamless unified virtual address space for CPU and GPU memory which greatly simplifies the GPU programming. Compute preemption is an important feature in Pascal architecture which allows the tasks to be preempted at instruction level granularity rather than the thread block granularity as in early architectures. GP100 SM ISA also provides new arithmetic operations which can perform FP16 operations very fast on a single processor CUDA core and also allows storage of two FP16 values in 32-bit GP100 registers. This allows fast and efficient training and deployment of large deep learning neural networks. GP100 also provides improved atomic operations. A comparison between the three NVIDIA GPU architectures is shown in Table 2. NVIDIA's next generation GPU is to be launched soon in early 2018.

Journal of Graphic Era University
Vol. 6, Issue 2, 207-223, 2018
ISSN: 0975-1416 (Print), 2456-4281 (Online)

Table 2. Comparison between three Tesla GPU architectures (NVIDIA Parallel Forall, 2014)

GPU	Kepler GK110	Maxwell GM200	Pascal GP100
Compute capability	3.5	5.2	6.0
Threads per warp	32	32	32
Max warp per multiprocessor	64	64	64
Max threads per multiprocessor	2048	2048	2048
Max thread blocks/multiprocessor	16	32	32
Max 32 bit registers per SM	65536	65536	65536
Max registers per block	65536	32768	65536
Max registers per thread	255	255	255
Max thread block size	1024	1024	1024
CUDA cores per SM	192	128	64
Number of SMs	8	16	60
Total CUDA cores	1536	2048	3840
Shared memory size/SM configurations	16K/32K/48K	96K (dedicated)	64KB(dedicated)
L1 cache/SM	64KB	64KB (split)	24KB(dedicated)
L2 cache	512KB	2048KB	4096KB

7. Conclusion

In this paper we have given a detailed r**e**view of the state of the art Fermi GPU architecture focusing on the programming model and the debugging tools. Key to performance improvement in CUDA applications is to reduce the global memory latency by providing massive multithreading so that the cores have enough amount of work to perform. Though porting any algorithms to the GPU is fairly easy fine tuning the programs to exploit the maximum capacity of the GPU is a major challenge. The GPU code has to be largely optimized to attain the maximum efficiency of the GPU being used. In this paper we have reviewed with examples some common programming strategies adopted by the GPU programmers to take maximum advantage of the GPU programming. We have also reviewed some common optimization techniques like kernel optimization, memory optimization, register optimization adopted by the GPU programmers to fine tune the GPU applications. Further, in this paper we have also given a brief review of some trends related to GPU computing. A brief review of the NVIDIA GPUs launched after the Fermi architecture and the new additional features included in each GPU is also discussed.

Our future work includes parallelization of some powerful image retrieval algorithms using the latest CUDA architecture and fine tune the application using the different programming and optimization strategies discussed. However, achieving peak achievable performance for a particular architecture rather than getting stuck in the local maximum of performance is a big challenge.

References

Bhangale, U., Durbha, S. S., King, R. L., Younan, N. H., & Vatsavai, R. (2017). High performance GPU computing based approaches for oil spill detection from multi-temporal remote sensing data. Remote Sensing of Environment, 202, 28-44.

Brodtkorb, A. R., Hagen, T. R., & Sætra, M. L. (2013). Graphics processing unit (GPU) programming strategies and trends in GPU computing. Journal of Parallel and Distributed Computing, 73(1), 4-13.

Chakroun, I., Mezmaz, M., Melab, N., & Bendjoudi, A. (2013). Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. Concurrency and Computation: Practice and Experience, 25(8), 1121-1136.

Chen, X., Wang, C., Tang, S., Yu, C., & Zou, Q. (2017). CMSA: a heterogeneous CPU/GPU computing system for multiple similar RNA/DNA sequence alignment. BMC Bioinformatics, 18(1), 315.

Domínguez, J. M., Barreiro, A.J.C. Crespo, O. García-Feal, & Gómez-Gesteira, M. (2016). Parallel CPU/GPU Computing for smoothed particle hydrodynamics models. In Recent Advances in Fluid Dynamics with Environmental Applications (pp. 477-491). Springer International Publishing.

Doulgerakis, M., Eggebrecht, A., Wojtkiewicz, S., Culver, J., & Dehghani, H. (2017). Toward real-time diffuse optical tomography: accelerating light propagation modeling employing parallel computing on GPU and CPU. Journal of Biomedical Optics, 22(12), 125001.

Dubey, S. P., Kini, N. G., Kumar, M. S., & Balaji, S. (2016). Ab initio protein structure prediction using GPU computing. Perspectives in Science, 8, 645-647.

Fauzia, N., Pouchet, L. N., & Sadayappan, P. (2015, February). Characterizing and enhancing global memory data coalescing on GPUs. In Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on (pp. 12-22). IEEE.

Jung, J., & Bae, D. (2018). Accelerating implicit integration in multi-body dynamics using GPU computing. Multibody System Dynamics, 42(2), 169-195.

Ke, J., Sowmya, A., Guo, Y., Bednarz, T., & Buckley, M. (2016, November). Efficient GPU computing framework of cloud filtering in remotely sensed image processing. In Digital Image Computing: Techniques and Applications (DICTA), 2016 International Conference on (pp. 1-8). IEEE.

Kim, K., Lee, S., Yoon, M. K., Koo, G., Ro, W. W., & Annavaram, M. (2016, March). Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on (pp. 163-175). IEEE.

Kimanius, D., Forsberg, B. O., Scheres, S. H., & Lindahl, E. (2016). Accelerated cryo-EM structure determination with parallelisation using GPUs in RELION-2. Elife, 5, e18722.

Kirk, D. B., & Wen-Mei, W. H. (2016). Programming massively parallel processors: a hands-on approach. Morgan Kaufmann. ISBN: 9780123914187, Elsevier.

Lee, D., Dinov, I., Dong, B., Gutman, B., Yanovsky, I., & Toga, A. W. (2012). CUDA optimization strategies for compute-and memory-bound neuroimaging algorithms. Computer Methods and Programs in Biomedicine, 106(3), 175-187.

Lee, S. Y., & Wu, C. J. (2014, March). Characterizing the latency hiding ability of gpus. In 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (pp. 145-146). IEEE.

Li, C., Yang, Y., Feng, M., Chakradhar, S., & Zhou, H. (2016, November). Optimizing memory efficiency for deep convolutional neural networks on GPUs. In High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for (pp. 633-644). IEEE.

Ma, Y., Chen, L., Liu, P., & Lu, K. (2016). Parallel programming templates for remote sensing image processing on GPU architectures: design and implementation. Computing, 98(1-2), 7-33.

Mantas, J. M., De la Asunción, M., & Castro, M. J. (2016). An introduction to GPU computing for numerical simulation. In Numerical Simulation in Physics and Engineering (pp. 219-251). Springer International Publishing.

Patterson, D. (2009). The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges. Nvidia Whitepaper, 47.

Siegel, J., Ributzka, J., & Li, X. (2011). CUDA memory optimizations for large data-structures in the gravit simulator. Journal of Algorithms & Computational Technology, 5(2), 341-362.

Smirnov, A. V. (2016). FIESTA4: Optimized Feynman integral calculations with GPU support. Computer Physics Communications, 204, 189-199.

Sundfeld, D., Havgaard, J. H., Gorodkin, J., & De Melo, A. C. (2017, March). CUDA-Sankoff: using GPU to accelerate the pairwise structural RNA alignment. In 2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) (pp. 295-302). IEEE.

Tan, G., Li, L., Triechle, S., Phillips, E., Bao, Y., & Sun, N. (2011, November). Fast implementation of DGEMM on Fermi GPU. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (p. 35). ACM.

Wen-Mei, W. H. (2011). GPU computing gems emerald edition. Elsevier.

Wittenbrink, C. M., Kilgariff, E., & Prabhu, A. (2011). Fermi GF100 GPU architecture. IEEE Micro, 31(2), 50-59.

Wu, B., Zhao, Z., Zhang, E. Z., Jiang, Y., & Shen, X. (2013, February). Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In ACM SIGPLAN Notices (Vol. 48, No. 8, pp. 57-68). ACM.

Wu, Y., Song, J., Ren, K., & Li, X. (2017). Research on Log GP based parallel computing model for CPU/GPU cluster. In Information Technology and Intelligent Transportation Systems (pp. 409-420). Springer International Publishing.

Zhang, E. Z., Jiang, Y., Guo, Z., & Shen, X. (2010, June). Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In Proceedings of the 24th ACM International Conference on Supercomputing (pp. 115-126). ACM.

Zhou, Q., Chen, W., Song, S., Gardner, J. R., Weinberger, K. Q., & Chen, Y. (2015, January). A reduction of the elastic net to support vector machines with an application to GPU computing. In AAAI (pp. 3210-3216).

Links-

Cornel Virtual Workshop (2013, July). Retrieved from https://cvw.cac.cornell.edu/gpu/

CUDA Optimization Techniques (2010). Retrieved from

CUDA-GDB (NVIDIA CUDA Debugger, 2010, October 14). Retrieved from

http://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/cuda-gdb.pdf

http://www.cs.virginia.edu/~mwb7w/cuda_support/optimization_techniques.html

https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

NVIDIA Maxwell GM204 Architecture whitepaper (2016, March 9). Retrieved from https://www.microway.com/download/whitepaper/NVIDIA_Maxwell_GM204_Architecture_Whitepaper.pdf

NVIDIA Parallel Forall: Five Things You Should Know About the New Maxwell GPU Architecture (2014, February 21). Retrieved from https://devblogs.nvidia.com/parallelforall/5-things-you-should-know-about-new-maxwell-gpu-architecture/

NVIDIA Tesla P100 whitepaper, (2016). Retrieved from

NVIDIA, C. (2010). Cuda programming guide, version 2.3. NVIDIA Corporation.

NVIDIA's Next Generation CUDA Architecture: Kepler GK110 (2013, Febrauary 8). Retrieved from https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Pan American Advanced Study Institute (PASI), Boston University (2011, January 31). Retrieved from www.bu.edu/pasi/materials/post-pasi-training/

S3478-Debugging CUDA kernel code with NVIDIA NSight Visual Studio Edition (2013, March 19). Retrieved from http://on-demand.gputechconf.com/gtc/2013/presentations/S3478-Debugging-CUDA-Kernel-Code.pdf