Accelerating Intensity Layer Based Pencil Filter Algorithm using CUDA

Dissertation

submitted in partial fulfillment of the requirements
for the degree of

Master of Technology, Computer Engineering

by

Amol S. Zagade
Roll No: 121022012

under the guidance of

Prof. V. S. Inamdar

Department of Computer Engineering and Information Technology
College of Engineering, Pune
Pune - 411005.

June 2012

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY, COLLEGE OF ENGINEERING, PUNE

CERTIFICATE

This is to certify that the dissertation titled

Accelerating Intensity Layer Based Pencil Filter Algorithm using CUDA

has been successfully completed

By

Amol S. Zagade
(121022012)

and is approved for the degree of

Master of Technology, Computer Engineering.

Prof. V. S. Inamdar,
Guide,
Department of Computer Engineering
and Information Technology,
College of Engineering, Pune,
Shivaji Nagar, Pune-411005.

Dr. Jibi Abraham,
Head,
Department of Computer Engineering
and Information Technology,
College of Engineering, Pune,
Shivaji Nagar, Pune-411005.

Date : _____

Abstract

Non photorealistic rendering is branch of image processing which tries to achieve artistic effects on images like painting, sketches etc. Pencil sketching is one of the important areas of non-photorealistic rendering. There are number of algorithms developed to simulate this effect. These algorithms give better visual effects but are time consuming. Therefore, these algorithms cannot be used for applications such as animation that requires faster outputs. There is need to develop faster algorithms or find out faster version similar algorithms. Area where these algorithms are applied such as algorithms, there, systems available are equipped with graphical processing units (GPU). With the computing power available onto the GPUs, these algorithms can be accelerated. This project report considered, intensity layer based pencil filter algorithm that accepts an image and generates pencil sketch for image. This algorithm is compute intensive and can be accelerated with the help of GPU. This project implements this algorithm on CUDA programming environment with GPU hardware as GeForce GTX 280. The results show maximum performance gain of 109 times than the conventional serial implementation of algorithm in terms of execution time.

Acknowledgments

I express my sincere gratitude towards my guide **Prof. Vandana Inamdar** for her constant help, encouragement and inspiration throughout the project work. Without her invaluable guidance, this work would never have been a successful one. I am extremely thankful to **Dr. Jibi Abraham** for providing me infrastructural facilities to work in, without which this work would not have been possible. Last but not least, I would like to thank my family and friends, who have been a source of encouragement and inspiration throughout the duration of the project.

Amol S. Zagade
College of Engineering, Pune

Contents

List of Figures

Chapter 1

Introduction

Non photo-realistic rendering (NPR) is branch of image processing which deals with producing artistic images. Achieving drawback of artists, making them irregular is an ultimate goal to make images feel like an art, not a computer generated image. Approaches for NPR are different from those of photo-realistic rendering (equations to solve to get the resulting renditions). The reason this area is being studied is, evidences that says non photo-realistic renditions tells much more and specific than photo realistic renditions. There are so many techniques discovered to address the non photo-realistic rendering areas. Irregularity mentioned earlier is must; otherwise it will result in synthetic and mechanical work which is hardly an artistic work.

Pencil Drawing is an area of non photo-realistic rendering which is studied a lot; so many techniques have been developed to produce pencil sketches from an image. These techniques take long time to generate image. Considering Frame rate requirements, to produce real time pencil sketches for applications like animation sequence generation, these methods will not work. There is need to accelerate these methods so that these methods can be used to produce real time pencil sketches. Computer hardware used in areas like animation and computer vision is usually equipped with graphical processing unit (GPU) which can be used to accelerate such applications (Tony DeRose in his interview). This report discusses the method for creating pencil sketches from the real images and accelerating this method with the help of graphical processing unit.

Graphics Processing Units (GPUs) are hardware devices available with large number of highly programmable components. GPUs evolved from fixed function pipeline to full-fledged general purpose programmable processing unit [3]. Initially, GPUs were only used for accelerating graphics rendering applications with the help of shading languages. Computer scientist thought of making use of GPUs for computation other than graphics rendering i.e. general purpose computation. This is done with the help of shading languages. Later, a programming environment which is specifically for general purpose computation on GPU is developed by NVIDIA named CUDA. It can utilize

abundance of computation power associated with GPUs for non-graphics computation. Computation other than graphical rendering done on GPUs is termed as "General Purpose Computation on Graphical processing unit (GPGPU)".

CUDA can be called as C for parallel computers. CUDA basically is a parallel computing architecture and programming model. It works on principle, write code for one thread and instantiate the same for multiple threads. This can be done in familiar programming languages like C. Advantage of CUDA architecture is once program is compiled for one processor, it can be executed on any CUDA supported hardware without recompiling. CUDA enables general purpose computing on GPU hardware, gives exposure to computational horse power of GPU. CUDA is a hardware software co-designed environment to achieve computational performance. With CUDA, supercomputing is available to larger number of users without spending much on HPC hardware. This report gives details about implementation of pencil drawing algorithm on CUDA and results showing performance gain in terms of speed up. Finally, conclusion and future work related to project.

Rest of the report is arranged in following order:

- Chapter 2 details literature survey.
- Chapter 3 gives information about GPU computing.
- Chapter 4 describes compute unified device architecture.
- Chapter 5 lists non photo-realistic rendering types and application.
- Chapter 6 describes intensity layer based pencil filter algorithm and its implementation on CUDA.
- Chapter 7 displays various pencil sketch images generated and lists speed up achieved on different image sizes.
- Chapter 8 concludes the report and gives future aspects of work.

Chapter 2

Literature Survey

According to surveyed literature, it is found that GPU computing is emerging as low cost alternative for high performance computing. GPUs can be used for general purpose computing with various programming environment available such as OpenCL, CUDA, DirectCompute, AMD Stream. OpenCL is an open source framework[4] developed by Khronos Group. It is developed for writing programs that execute across different platforms including CPU, GPU and other processors. It is available with programming language which is used to write functions for GPUs and APIs to manage the hardware devices. CUDA is most popular development environment[5]. CUDA is parallel computing architecture developed by NVIDIA for their GPUs. CUDA programming is easy to adapt, as it is developed with little extensions to basic and familiar programming languages like C. There are various libraries available in CUDA. CUDA is a computing system available to programmers in various industry standard programming languages. There are large number of libraries available in CUDA. Performance improvement achieved with CUDA is scalable as per the computing capability of GPUs. DirectCompute and AMD Stream are similar programming environments designed to utilize computing capabilities available on GPUs designed by Microsoft and AMD respectively.

Image processing application exhibits a lot of parallelism[6]. Non photo-realistic rendering is an area of image processing which deals with producing artistic images which hardly look like photo-realistic image but are consisting of necessary details which conveys necessary description in easy way[7, 8]. Large number of algorithms developed in this area, but these algorithm are compute intensive. They tend to be slow and needs a faster versions to apply those in real life scenarios. Pencil sketch drawing is one of the areas of non photo-realistic rendering.

Pencil sketch drawing is a popular art which is used extensively. In general, a pencil sketch is a drawing in which objects are depicted in darker color like pencil on a light background (white) with variation in the shades of darker color [9]. Pencil sketch

generation technique is studied a lot. Large number of algorithms have been developed to simulate pencil sketch generation from photographs. We have gone through various research papers to find out best suitable method which gives better effects and can be parallelized. Jin Zou [9] proposed a four step method to simulate pencil drawing. It detects borders of the objects in image and color borders according to local gradients of pixels. A gradient transform is used to get closer image to final output. Final, step is for smoothing the pencil sketch drawing, as result of previous step left the contours of image non continuous.

Pencil filter algorithm proposed by Mao [10] is based on Line Integral Convolution (LIC). Line integral convolution is a vector field visualization technique. Mao used LIC to simulate pencil strokes, only case here is all vectors are in one direction. In his approach, he first generated white noise image from the given input image, to match tone with output pencil sketch, divided the image into regions and find boundaries of regions. LIC accepts white noise and vector field to produce an image, with each pixel stretched in the direction of vector field that passes through that point in both direction. LIC applied to each region separately. To generate output image they[10] mixed output after applying LIC for all the regions, by highlighting the boundaries.

Based on Mao's approach [10], Yamamoto proposed enhanced LIC pencil filter algorithm, which also uses similar technique to produce pencil strokes. In his approach, instead of producing pencil sketch using different regions separately, he considered different layers of complete image to produce final sketch. It divides the images into different layers of intensity. He applied LIC in same way as in Pencil filter by Mao [10]. For each layer, he applied LIC in different direction but all vectors in one application are in same direction. Finally, to generate final output add all layers after applying LIC and highlight boundaries. This algorithm is faster than basic pencil filter algorithm as it avoids segmentation and produces similar results and better in case of some inputs[11].

We chose Yamamoto's approach[11] to accelerate using CUDA. As it can be parallelized and produces better pencil sketch images than other methods.

Chapter 3

GPU Computing

Microprocessors with single Central Processing Unit(CPU), like Intel Pentium family and AMD's Opteron family, made tremendous improvement in performance increase and cost reductions of computer applications for around twenty years. They make, giga-floating point operations per second possible on the desktop and hundreds of giga-floating point operations on cluster servers[1]. This continuous growth in performance keep, software developers to create applications with better user-interfaces, and generate more useful results. The computer users, in turn, ask for more improvements as they will be habitual to these improvements, causing need of growth for computer industry.

During this period, software developers relied on advancements in hardware to make their applications faster; with advent of newer hardware same software runs faster on previous version. However, from 2003 due to energy consumption and heat dissipation issues, further increase in clock frequency and number of activities that can be completed in single clock cycle, has stopped[1]. Causing all microprocessor vendors switched to multiple processors on a single chip, are used as processing units, recently. These new processing units made a huge impact on software developer. They need rigorous programming to get benefits from computing devices arranged in this fashion.

Traditionally, almost all softwares were developed as sequential steps, as described by Von Neumann in 1945. The execution of these softwares can be understood by going through all steps sequentially. It was possible to achieve higher speed versions of same code, when these softwares are executed on newer machines with higher clock speeds. But, this trend no longer exist, as existing softwares were only execute on one of the core out of many available causes lesser speed up. This happened, because clock speed of cores does not increased a lot over later versions instead processing cores on a single chip increased. It left software programmers reluctant to add newer features and capabilities to softwares. Such a microprocessor architecture causes, modified version of softwares to be designed to utilize the processing power available on microprocessor chips. It takes tremendous programming efforts to make it happen.

Application software, that will achieve higher speed up with newer version of microprocessors, must contain parallel programs. These parallel programs will consist of multiple threads that can execute in parallel in cooperation on different processing cores of same chip. This programming architecture is not new, research scientist are working on this for decades. The difference here is, they were working on larger compute intensive tasks but transformation in microprocessor architecture causes sudden increase number of application that must be developed in parallel.

3.1 GPU as part of mainstream computer

Energy consumption and Heat dissipation causes microprocessor designer, to take two different approaches, multi-core and many-core. Multi-core designs tries to maintain execution speeds of sequential programs while making them execute on multi cores. The number of cores in this architecture doubles with every generation. In contrast, many-core approach focuses on executing parallel processes. These designs started with large number of small sized cores and count doubling with every generation. An example of this design, is NVIDIA's GeForce GTX 280 Graphical Processing units with 240 cores, each shares its instruction and data cache with seven others. Many-core processors mainly GPUs led the race for giga-floating point performance since 2003,as illustrated in Figure 3.1. While general purpose processors does not see any larger growth in its performance. The ratio between many-core GPUs and multi-core processors, of performance throughput is of 10 to 1[1]. Till 2009, Raw speed that these processors support are around 1000 Gigaflops for GPUs versus 100 Gigaflops for multi-core CPUs.

This large performance gap between multi-core and many-core GPUs motivates application developers to move their compute intensive tasks of their application to execute on GPUs. It gives opportunity to developers to add additional features and graphical user interfaces. It is natural that, more the compute intensive part of application, more it has scope for parallelization.

This large amount of gap between the many-core GPUs and general purpose CPUs is because of their design thinking. General purpose CPUs were optimized for sequential programs. It makes use of well designed control logic to execute instructions from a single thread in parallel or even out of order of sequence while maintaining appearance of sequential execution. Large cache is provided to reduce instruction and data access latencies. But, none of control logic and cache contribute in peak computation speed. Memory bandwidth is an important issue need to be considered while comparing these two. Graphics processors have around 10 times the bandwidth of CPU chips. The way Legacy operating systems, application softwares and I/0 devices memory access work it

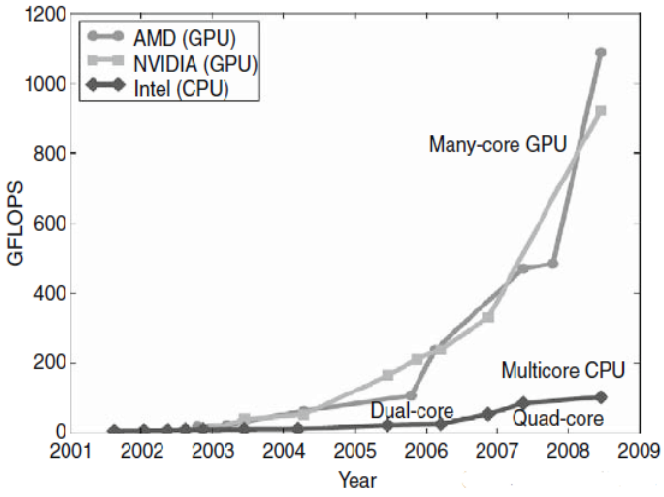

Figure 3.1: Comparison of GPU Vs CPU (GFlops) over the years [1]

becomes difficult to increase the memory bandwidth of CPU chips. In contrast, GPU system can easily increase the bandwidth. NVIDIA's GTX GPU has peak bandwidth of 200 GB/s whereas CPUs has it around 30 GB/s. CPUs will suffer from lesser bandwidth for quite few years.

The need of such a design was due to exertion of economic pressure, for ability to achieve large number of floating point operations per seconds per video frame for gaming environment. This will lead to, maximize chip area and power budget for floating point calculations. This architecture take advantage of large number of threads available. It finds work to do when threads are busy waiting for long latency memory accesses. Processing cores are associated with small cache to avoid memory access latency. This design dedicates larger portion of processor chip for processing units rather than control and cache.

GPUs are designed as numerical computing engines, and it cannot perform some tasks that CPU can perform very well. It is like, the serial portion of application is executed on CPU and whenever parallel compute intensive portion is occurred, it is given to GPU. This is why NVIDIA introduced Compute Unified Device Architecture (CUDA), which supports execution of application on CPU/GPU hybrid combination.

While developing software application, performance is not the only factor that decides the processors for running their application. There is need to consider other factors, too. These factors are market presence of underlying processor and second is development cost for application. Whenever a development firm thinks of developing any software application, it considers amount of targeted customers. If application needs

some specific hardware requirements then there is huge decline in customer base. This has been a critical problem with parallel applications, very few applications funded by government and larger organizations are developed successfully. This thing has changed with GPUs, because of its huge customer base. Since launch of CUDA in 2007, over 200 million GPUs have been installed[12]. This makes parallel computing possible with mass market product. Such a large market, makes application developers get attracted towards GPUs.

Apart from all above reasons, Practical form factor and accessibility are important while deciding applications architecture. It is because, supercomputing clusters are larger in size, physically. It makes these machines not suitable for actual deployment with applications. In contrast, it is possible with GPUs. Similarly, Applications with need of higher accuracy, must need processing units those support IEEE floating point standard. Earlier versions of GPU does not supports this standard but later versions do support strong support for these standards(single precision). While double precision is supported at around half the number of floating point operations per second than single precision[1].

3.2 Architecture of Modern GPU

Fig 3.2. shows architecture of CUDA-capable GPUs. It is designed as array of threaded streaming multiprocessors (SMs). In figure only two SMs form a block but this varies with versions of GPUs. Each SM consist of number of streaming processors(SPs) that share control logic and instruction cache. Now, each GPU comes with graphics double data rate (GDDR) DRAM, which is called as *global memory* as shown in figure3.2. This memory differ from the one in CPU. It holds video images and texture information for 3D rendering, when used for graphics application. Whereas in numerical computing it acts as high bandwidth off-chip memory, with somewhat latency to system memory. Effect of latency can be reduced effectively with high bandwidth.

nVIDIA's G80 GPU that introduced CUDA, is available with memory bandwidth of 86.4 GB/s whereas it is connected to CPU with communication bandwidth of 8 GB/s. It can transfer 4 GB/s data from CPU to GPU at the same time same amount of data can be transferred from GPU to CPU. When compared to memory bandwidth of GPU, it seems performance bottleneck. But it is comparable with CPUs memory bandwidth. Growth is expected with growth in CPUs memory bandwidth.

G80 GPU has 128 SPs (16 SMs with 8 SP each) with peak processing capability of around 500 Gflops. Each SP has multiply-add unit and one additional multiply unit. In addition, it has one special function unit performing floating point unit performing

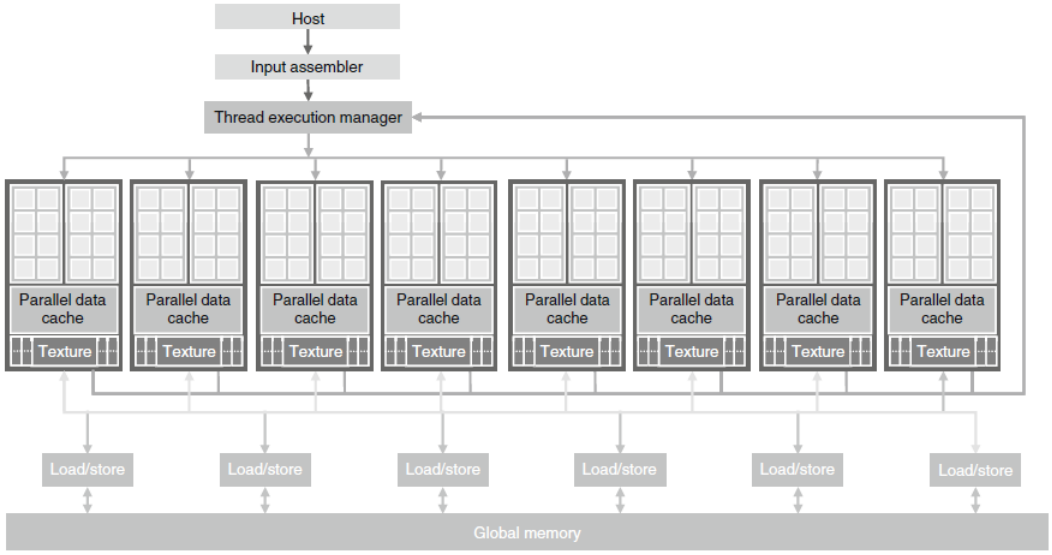

Figure 3.2: Modern GPU Architecture [1]

operations such as square root and transcendental functions. GT 200 with 240 SPs gives more than 1 Teraflops. SPs are massively multi threaded makes it capable of running thousands of threads on each, per application. G80 and GT 200 can span around 12000 and 30000 thread at a single time instant. It is clear that GPUs thread carrying capacity is increasing rapidly. It is now left with application developers, how to utilize the such a high processing capability.

3.3 Programming GPUs

Various parallel programming models have been proposed in last few years. MPI for scalable cluster computing and openMP for shared memory based multiprocessor systems are the most widely used among all. MPI is a model used where no memory to share as a communication media instead explicit message passing is done for synchronization. MPI is a huge success in scientific computing. It can run on the systems with more than 1,00,000 nodes. Porting application on MPI needs extreme efforts because of absence of shared memory across computing nodes. On the other hand, provides shared memory for parallel execution which makes it advantageous over MPI. Currently, CUDA provides shared memory which is less and Programmer needs to manage all shared memory between CPU and GPU. This feature is is similar to "one sided message passing", absence of same considered as major disadvantage of MPI[1].

OpenMP provides shared memory architecture taking advantage over MPI. However,

is does not scale beyond some hundred nodes due to issues with respect to thread management and cache coherency. In contrast, CUDA is highly scalable with simple, low overhead of thread management and no cache coherency hardware requirements. Many applications fit into the programming models of CUDA utilizing computing capability.

Programming constructs of OpenMP and MPI are very much similar to CUDA, although openMP's compilers provide automated management of parallel execution. Researchers are working adding automation on parallel execution management and performance optimization with CUDA. Developers with expertize in openMP and MPI will find CUDA easy to learn.

Major companies, Apple, Intel, AMD/ATI and NVIDIA came together and developed a programming model, called OpenCL similar to CUDA. OpenCL provides language extensions and programming APIs to programmers, to manage parallelism and data transfer on these parallel processors. OpenCL is standardized programming model when an application is programmed on it. It can work on some other processors without any modifications.

3.4 History of GPU Computing

While Developing CUDA and OpenCL applications, programmers look at GPUs as massively parallel computing processors programmed with simple programming languages. To program these processors there is no need of knowing the graphics algorithms and terminology. However knowing the details of transformations of these processors, gives idea about pros and cons about the processors. It helps understanding the designs of systems with massive multi threading, smaller cache sizes and higher bandwidth comparing to CPUs. This study will also helps in understanding the details of further versions of such systems.

3.4.1 Computing with fixed function graphics pipelines

Graphics pipeline hardware evolved from large systems to small workstations then PC accelerators. During this period, these graphics subsystems decreased in price from $50,000 to $200. During same period, the performance increased from 50 million pixels per second to 1 billion pixels and 1,00,000 vertices per second to 10 million vertices per second [13]. Although this advancements are due to the shrinking sizes of the semiconductor devices, there is also improvements in graphics algorithms and hardware design increased hardware capabilities.

Till late 90's, leading performance graphics hardware was configurable but not pro-

grammable. In similar span, application programming interface(API) libraries became popular. These API's can allow a gaming application send graphics processing units to draw objects onto the display device. DirectX API is one such API is for media functionality. Direct3D is component of DirectX, provides interface functions to graphics processors. Similar API is OpenGL developed by open source community and is popular in professional workstation application.

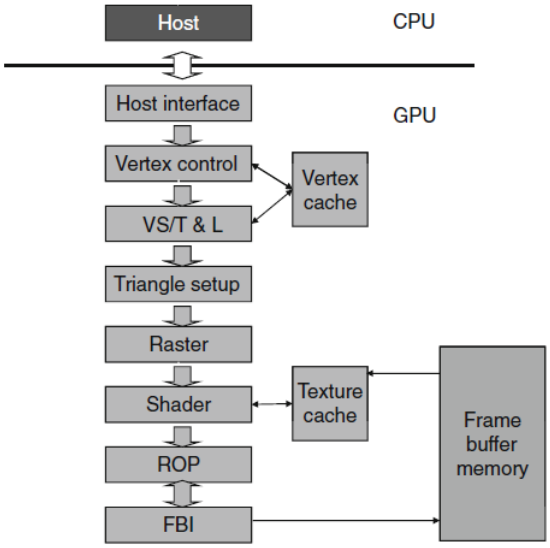

Figure 3.3: Fixed Function Graphics Pipeline [1]

Fig. 3.3 gives an example of fixed function graphics pipeline in NVIDIA's GPUs. Input given to this pipeline is commands and data from CPU. Host interface typically contains a specialized direct memory access (DMA) hardware to efficiently transfer data to and from host memory to graphic pipeline. The host interface returns the status and result data back to CPUs.

In a graphics application, vertex is corner of polygon which as a component is rendered. nVIDIA's graphics cards renders triangles so for this case vertex is corner of triangle. Any object is drawn as set of triangles. The size of triangles decides quality of rendered output. Finer triangles gives quality rendering but also cost for larger computing. Vertex control stage accepts the triangle data and make it available for further processing by converting it to a form understandable to graphics hardware.

Vertex, Shading, transform and lighting (VS/ T&L) stage in fig. 3.3 processes the vertices and gives values to the vertices considering colors, normals, tangent and texture coordinates. Actually color to triangle pixels are assigned later in further stages. Trian-

gle set up stage generates equations to further determine the colors of pixels touched by triangles. The raster stage computes the pixels inside of triangles and further per-vertex values for these pixels. Finally, shader stage determines color for each pixel to be rendered. This stage results are combined efforts of interpolation of texture colors, texture mapping and per-pixel lighting mathematics reflections etc. These technique helps in making rendered image realistic. Final raster operation is for color raster and tries to work for anti-alizing and transparency. FBI manages memory accesses of display frame memory.

3.4.2 Programmable Unified graphics processors

In 2001, NVIDIA's GPUs from GeForce series, provided capability to program private VS/ T&L stage. Further, GPUs extended programmability and floating point capability pixel shader stage and also made texture memory accessible from vertex shader. This programming capability added to shader, was for making functionality of different stages as generic, for programmers.

Certain stages of graphics pipeline perform lot of floating point calculations such as generating pixel colors on completely independent data. This is the major factor while designing CPU and GPU. These GPUs can render a frame around $1/60^{th}$ of a second with a million triangles and 6 million pixels. It left with good option to parallelizing these computations.

In 2006, GeForce 8800 GPU gave different stages of graphics pipeline to array of unified processors. Actually this pipeline is a loop for much fixed-function graphics functions for vertex shading, as shown in fig 3.4 , geometry processing and pixel processing each at once. Rendering algorithm has vast change in loads of these stages, unification allows dynamic allocation of processing resources to different stages and make proper utilization of them.

From fig. 3.4, it is clear that earlier GPUs used different set of processors for different stages in a pipeline. Whereas in unified GPUs, all stages uses same set of processor arrays and these processors are divided dynamically as per the load for each stage of processors. Result of previous stage of shading is fed to same processors for next stage. This way proper load balancing is done on unified GPUs.

3.5 GPGPU: Programming GPU for general purpose

GPU designs moved towards more unified processors from fixed-function pipelines making them similar to HPC parallel computers. Researchers took note of performance

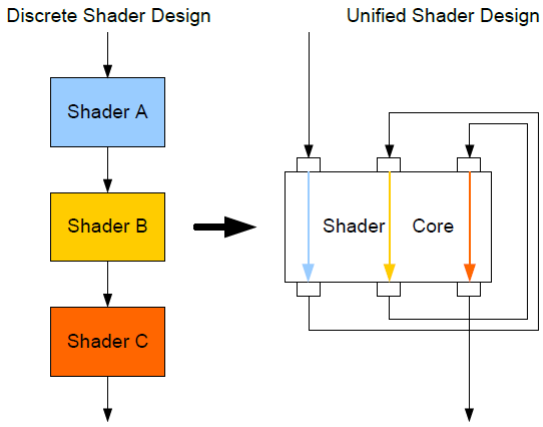

Figure 3.4: Differnce between execution of stages in unified GPUs [1]

improvement in GPUs, thought of utilizing this power for compute intensive scientific and engineering applications. But DirectX 9 APIs were able solve only problems related to graphics applications because it was designed like that. To make use of these resources, there was need to mold the problem into a problem that look like graphics problem. Then problem could be launched with DirectX 9 and OpenGL. The input to application is stored onto texture memory. It is given to processing units in the form of triangles. This system results output as a pixels after rasterization.

GPU processors and Buffer memory are designed for graphics applications not for any other general computations. The result of application fed to GPUs are pixel color values which indicates graphics processors has very limited memory access capability. Programs for shading uses texture memory to store its input data and to write output data, they does not calculate any specific address. This result, pixel color value, is can be stored in two dimensional frame buffer memory.

To design an application, where result of one stage is given to next, was done by writing results of previous stage to pixel frame buffer and use this buffer as input to the pixel fragment shader for next stage. This computations are done without any specific user defined data types, data was stored in one-, two- , or four- component vector arrays. This whole process of molding application to graphics application was tedious and need rigorous programming efforts. This process of using GPUs for scientific and engineering computations is termed as “GPGPU”.

NVIDIA figured out, how useful this GPUs can be for computations other than graphics and programmers can think of them as a high performance processors. They started out work on high efficiency of floating point and integer processor that can be

utilized for workloads in graphics pipelines. Newer GPUs are available with instruction memory, instruction cache and instruction sequencing control logic. This cost of adding hardware was nulled by multiple shader processors which shares instruction cache and sequencing control logic. This is because same program is needed for multiple pixels. GPGPU programmers Tesla architecture provides parallel threads, barrier synchronization, and atomic operations to dispatch and manage parallel computing work. In 2006, NVIDIA developed CUDA to use these GPUs efficiently for general purpose computations. It has C/C++ compiler, libraries and runtime software to program application based on CUDA. With CUDA, programmers relieved of molding their applications in graphics APIs. This report discusses CUDA in detail in Chapter 4.

Chapter 4

Compute Unified Device Architecture (CUDA)

As described in earlier chapter, CUDA is a programming environment using which we can program GPUs for general purpose computation. CUDA programmer sees execution system as a host and one or more devices. Host is a traditional CPU like Intel architecture based microprocessor in personal computer and device is Graphical Processing Unit (GPU) with massively parallel processors.

Todays software applications process large amount of data and takes long execution times when executed on multi core processors. This processing is done on independent data, it can be processed independently and simultaneously. It gives a lot of scope for parallelism in these applications.

4.1 Introduction

4.1.1 CUDA Program Structure

A CUDA program is a serial program with parallel sections. This CPU and GPU code is unified, it starts its execution on CPU with serial code and whenever a parallel code is occurred it is executed onto GPU and control is returned back to CPU when parallel code finishes execution, as shown in fig. 4.1. The NVIDIA CUDA compiler (nvcc) separates two different codes(host and device) and compiles them separately. Host is a simple C code which is compiled with normal C compiler. Device code is different from host code and is programmed in programming language with some little extensions to host programming language. It is compiled with nvcc. A data parallel functions that executes on GPU are called as "*kernels*".

Kernel functions generates large number of threads to achieve large data parallelism

which are executed by GPU's processor array. For e.g. In the matrix multiplication, all calculations are done onto GPU, one thread can be created for each output element of matrix element i.e. 100×100 size matrix multiplication will have 10000 threads in total. CUDA threads are much lighter than the CPU threads. It takes very few clock cycles to create threads and schedule with hardware support, whereas CPU thread takes thousand of clock cycle to generate and schedule them.

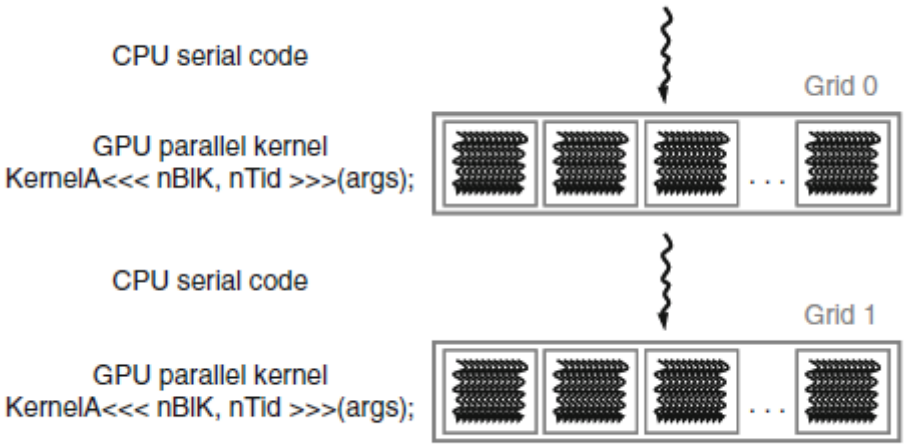

Figure 4.1: CUDA execution [1]

The execution of CUDA programs can be seen in fig. 4.1. GPU creates large number of threads, all threads are created in hierarchy called as "*grid*".

4.1.2 Device memories and data transfer

Host and devices has different memory spaces [12]. GPU devices has its own dynamic random access memory. To execute kernel functions onto the GPU devices, programmer must allocate memory and copy data from CPU memory to GPU memory to make it available for computing threads on GPU. After the execution, programmer needs to free memory allocated onto the device which does not require any more. CUDA runtime has APIs which can be used for these tasks like memory allocation and deallocation which are mentioned in detail in further sections.

Fig. 4.2 gives information about CUDA memory model and need of allocating, transfer and usage of different memories available on GPUs. It has Global memory and constant memory where programmer can transfer data from host device. Constant memory is a read only memory accessible to device code. Global memory is largest

memory available onto the GPUs. To make efficient use of different memories available, CUDA is provided with APIs to handle these memories. Fig 4.2 shows details about memory allocation and deallocation APIs. The API cudaMalloc() can be called from host machine to reserve memory onto the device. Host based malloc for CPU memory in C and cudaMalloc() in GPU have similar parameters to make CUDA with smaller extensions to basic programming language.

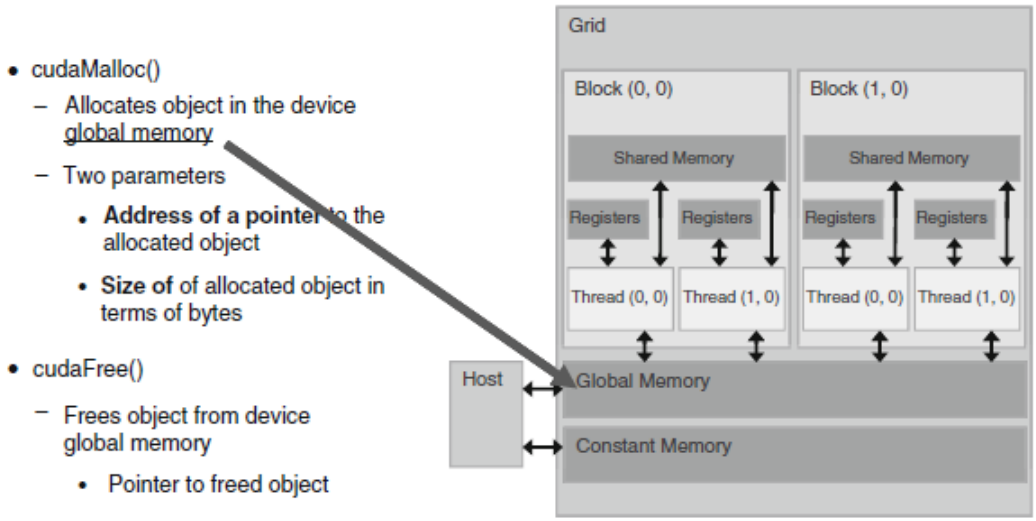

Figure 4.2: CUDA Memory Allocation and Deallocation [1]

First Parameter for cudaMalloc() function is address of a pointer where allocated memory's starting address will be stored and this address should be casted to (void **), as it should be generic to allow any type of objects and not any specific type. The second parameter is similar to its C version malloc, it is mainly for specifying how many bytes of memory to be actually allocated. A subsequent sample of code gives idea of how this allocation is done. Parameters given in code are address of a pointer variable of type float which is casted to (void **) and number of bytes to allocate are given in variable size. Further, cudaFree() is used to deallocate memory allocated. It takes pointer to memory location to deallocate as its only argument.

```
float *Md
int size = Width * Width * sizeof(float);
cudaMalloc((void**)&Md, size);
...
cudaFree(Md);
```

After allocation of memory to onto device, we can transfer data from CPU to this memory locations using CUDA API cudaMemcpy() and is shown in fig. 4.3. It takes 4 parameters- pointers to destination location, pointer to source location, number of bytes to be copied and specification of memory involved in coping operation as shown in fig. 4.3. Last parameter has four different values from host to host memory, host to device memory, device to host memory and device to device memory. This cannot be used to copy data between different GPUs.

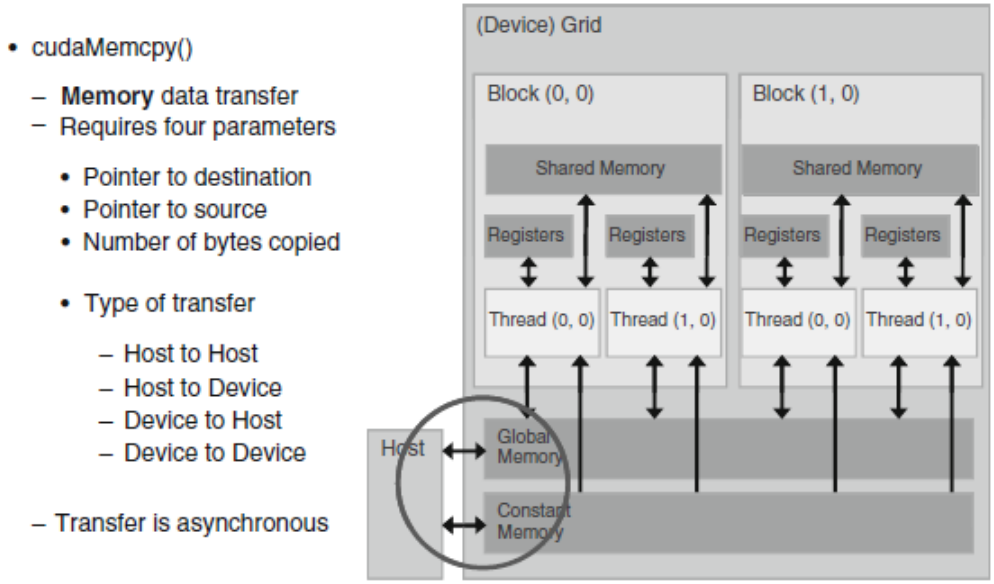

Figure 4.3: coping data from CPU memory to GPU memory [1]

4.1.3 Kernel functions and threading

```
//Thread specification of kernel
__global__ void sample_kernel(float *A, float *B, float *C)
{
// calculate thread identifier for specific memory access
int thread_id = threadIdx.x + blocIdx.x * blockDim.x;

C[ thread_id ] = A[ thread_id ] + B[ thread_id ];
}
```

Kernel functions are basic building block for parallel computation on to GPU. Kernel is a code which all threads executes on GPU, as all threads executes same code

	Executed on	Only callable from
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

Table 4.1: CUDA extension to C function declaration

programming this is an single instruction multiple data(SIMD) programming. Sample code structure mentioned above gives idea about how kernels are coded. The keyword __global__ in front of keyword void specifies that this function is not ordinary function but a kernel function which executes on graphical processing units. This function can only be called from the host and not from the device. Similar to this keyword, there are some other which adds meaning to function declaration are listed in table 4.1.3. These keywords are little extensions to basic function declaration methodology.

The functions with keyword __device__ is a device function i.e. it is executed onto the device and can only be called from kernel function or some other device function. Device functions cannot have recursive function calls and cannot be called with pointers to function. Similarly, keyword __host__ is used to specify that kernel is a host function and are called and executed on to host. These functions are simply C functions. Keywords __host__ and __device__ can be placed ahead of a function at the same time in any order, compiler will generate two versions of output code. One can be called and executed from device or kernel function and other only from host.

To differentiate among threads, CUDA added some keywords which specifies details about a thread. threadIdx.x and threadIdx.y are the thread co-ordinates. These keywords accesses hardware register to give details about the co-ordinates. When a kernel is launched, it is launched as a grid of threads. Where threads are arranged in blocks of threads, with each block containing threads arranged in one, two or three dimensions. Each block has same arrangement and same number of threads. Block co-ordinates of a threads are identified with blocksIdx.x and blockIdx.y. Maximum number of blocks in a grid are identified with gridDim.x and gridDim.y. Similarly, a thread in a block is identified with threadIdx.x, threadIdx.y and threadIdx.z. Maximum number of threads in a block are identified with blockDim.x, and blockDim.y. This thread organization can be well understood with fig. 4.4.

4.2 CUDA Threads

High performance with CUDA is achieved using fine grained parallelism by executing thousands of threads in parallel. CUDA launch of kernel, creates thousands of threads runtime and each thread executes same code on different set of data.

4.2.1 CUDA thread organization

As all threads created with kernel launch executes same kernel code, these threads depend on identifiers to distinguish among themselves and same identifiers are used to access different data sets during the process. Threads are arranged into two levels of hierarchy, and are identified with blockIdx and threadIdx. blockIdx is block identifier in hierarchy and threadIdx is a thread identifier. These variables gets their values assigned by CUDA runtime system. Kernel function has access to these variables and these variables return co-ordinates of threads. There are two more such variables for dimensions of block and grid, gridDim and blockDim. They return dimension of grid and dimension of block respectively.

Fig 4.4 gives details about these variables. It shows a grid containing N thread blocks, each with block identifier from 0 ot N-1. Each block has M threads with identifiers from 0 to M-1 . In this example, blocks and grids are arranged in one dimension. This grid will contains in all N * M threads.

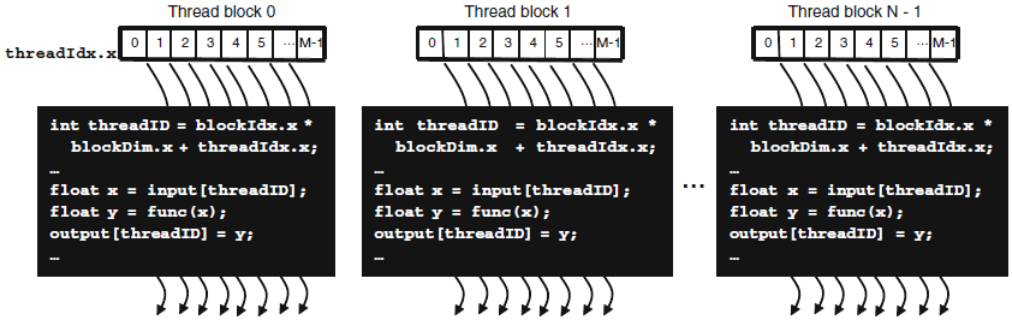

Figure 4.4: CUDA Thread Organization [1]

The code in black in figure is executed by that block(actually all blocks executes same code). A unique identifier to a thread is assigned using $threadID = blockIdx.x * blockDim.x + threadIdx.x$ statement, to differentiate among the data values to be accessed. Thread of block 0 has value for threaID as 0*M + 4 = 4. Similarly, thread 5 of block 4 has value for threadID as 4*M + 5.

Consider a example, a grid has 128 blocks (M =128) and each block contains 32 threads (N =32). Call to gridDim.x and blockDim.x will return 128 and 32 respectively. In total $128 * 32 = 4096$ threads created when kernel is launched and each has a different identifier. In fig. 4.4 shows code access input array and output array. Considering these arrays are declared with 4096 elements then each of these threads will access one

element each of input array and stores output value to output array at same index.

A grid can have blocks arranged in two dimensions or less and threads in block arranged in three dimensions or less. When a kernel is launched, its launching parameters decides how the blocks and threads in a blocks will be arranged. This is done with the help of c structure dim3, which has 3 parameters x, y and z , one for each dimension. As blocks in grid are only has two dimensions, third parameter in struct dim3 is ignored but it is good practice to keep it 1 for sake of clarity. The following code gives idea for how blocks shown in fig. 4.4 must have launched.

```
dim3 dimGrid(128,1,1);
dim3 dimBlock(32,1,1);
KernelFunction<<<dimGrid,dimBlock>>>(..);
```

Two declarations in the line one and two are for initializations of execution configuration. Both grid and block are one dimensional, therefore except first parameter rest are 1. Third statement is call to kernel with execution configuration variables. It indicates kernel is launched with 128 thread blocks and each block containing 32 threads each. Execution configuration are placed between <<< and >>> symbols, as shown in third line of code. It is possible to launch kernel with same set of configuration using following code.

```
KernelFunction<<<128,32>>>(..);
```

Maximum value of gridDim.x and gridDim.y is 65,535. These values are set at the time of kernel launch and these values cannot be changed when kernel is launched. All threads in a block has same values for blockIdx. Value for blockId.x ranges from 0 to gridDim.x-1 and for blockIdx.y from 0 to gridDim.y-1. Block is a collection of thread arranged as three dimensions. Each threadIdx has three components: threadIdx.x is x component, threadIdx.y is y component and threadIdx.z is z component. Values for threadIdx.x ranges from 0 to blockDim.x and in similar manner for threadIdx.y and threadIdx.z. Maximum value for each is one less than as specified in execution configuration. Block size is limited to 512. This means that multiplication of three components of threadIdx should be less than or equal to 512, with different combinations of threadIdx components possible. For e.g (512,1,1), (8,16,2) and (16,16,2) are allowed but (32,32,1) is 1024 which is not allowed as it exceeds 512.

4.2.2 Synchronization and transparent scalability

Threads those are in the same block coordinate their execution using barrier synchronization function, __syncthreads(). When a thread in kernel function calls __syncthreads(), its execution is halted till all threads in that block calls same function. This is to make sure that all threads in the block has completed certain portion of the code and are ready to execute next portion. No thread has started executing next portion of code.

Barrier synchronization is very simple and powerful method to coordinate parallel actions [1]. This we can relate with the scenario in real life. Consider, four friends go to a market in a car. They park their car in parking lot. They have different things to buy. It is more time consuming if they stayed together and purchased everyones goods one by one. Instead, if they purchase their things on their own by splitting from the group, they can do this in parallel and can save time. But to coordinate the activity, after purchasing they must wait for others at car or some already decided place. Otherwise, one or more persons will be left in the market.

The barrier synchronization __syncthreads() must be executed by all threads. When a __syncthreads() is placed in an if statement, then either all threads in a block execute the path that includes __syncthreads() or none of them should. In if-then-else statement, if both if and else path has __syncthreads() statement, then either all threads in a block execute if part or all of them should execute else part. CUDA will consider these two as different synchronization points. If some of them execute if part and some end up doing else part then they will wait for each other forever.

Barrier synchronization makes restriction on threads in a block. As all threads in a block wait for remaining threads to reach synchronization point, wait time should be time bounded. It should not happen that waiting time for remaining threads to reach barrier point is excessively high. CUDA runtime system makes sure that waiting time is not too long, by assigning all threads in a block to a same execution resource [1]. This guarantees that all threads in block finishes their execution in close proximity.

Now, it is clear that different blocks in a thread cannot be barrier synchronized. CUDA can execute different blocks in any order it wants beacause there is no dependency among the blocks as they do not wait for each other for finishing their task [1]. This allows us scale flexibly as the GPU system changes as shown in fig. 4.5. An application with low cost implementation with lesser execution units, will execute small number of blocks at a time. Whereas a high-cost implementation will able to execute more number of blocks simultaneously than low cost implementation. Ability to execute same code on different configuration at varying speeds makes it possible implement it for wide set of applications based on cost, power and performance requirements. This

feature to execute same code on different number of processing resources without making any changes is known as transparent scalability.

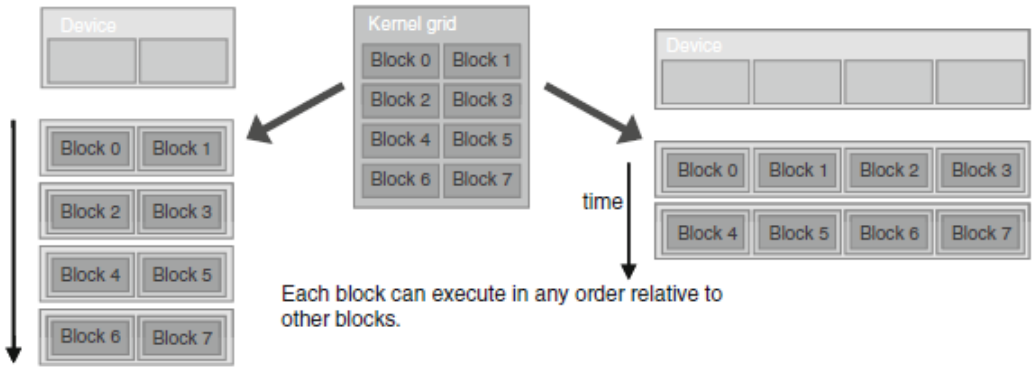

Figure 4.5: Transparent Scalability [1]

4.2.3 Transparent Scalability

CUDA generates grid of threads when a call to a kernel is made. These threads are assigned to a resources on a block by block basis. Execution resources are arranged as an array streaming processor; for e.g. NVIDIA GT200 has 30 streaming processors, 2 of them are shown in the fig. 4.6. Maximum 8 blocks can be assigned to each SM but only if sufficient number of resources to satisfy all the blocks. In situations where resources are not sufficient to make all eight blocks satisfy, then CUDA runtime system will make sure that lesser number of blocks are assigned to each SMs. With 30 SMs available GT 200 can accommodate maximum 240 blocks of threads at a time. But larger application can contain more than 240 blocks. In this case, runtime system makes a list of blocks and assigns new blocks of threads when SM finishes executing previously assigned thread blocks.

One of the main limitations of SM resource is number of threads assigned to a thread block and it could decrease the number of blocks assigned to a SM. Fig. 4.7 shows three thread blocks are assigned a SM. To maintain thread and block identifiers and track threads execution there is need of hardware resources which limits number of threads assigned to a SM in GT 200 design to 1024. These threads can be assigned as 4 blocks of 256 threads each or 8 blocks of 128 threads each but it is not possible to assign 16 blocks of 64 threads as this exceeds the limit for maximum number of blocks assigned to any SM of GT 200. As GT 200 has 30 SMs it can have 30,720 threads simultaneously assigned in SMs. If we execute same code that is written for GT 200 GPU on G80 which has lesser processing units, it will work fine but will take some more time. This feature

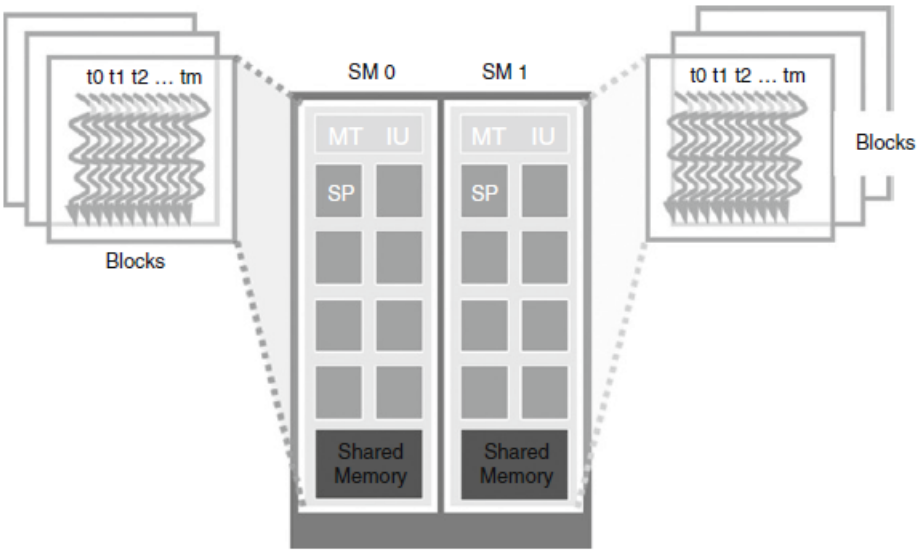

Figure 4.6: Streaming Processors in GT 200 [1]

is transparent scalability.

4.2.4 Thread Scheduling and Latency Tolerance

In GT 200 implementation, if a block is assigned to a SM, it is divided in 32-thread unit called as warps. The size of warp may vary from implementation to implementation. Warp is not a part of CUDA specification, but knowledge of its working helps in understanding execution and performance tuning of application on particular devices. The warp is a unit of thread scheduling in SMs. The fig. 4.7 gives details about division of block into warps for GT 200. Each warp consists of 32 threads of continuous threadIdx values. Threads 0 to 31 form first warp and 32 to 63 form second and so on. From number of blocks assigned to an SM and number of threads in a block we can find out number of warps that reside in each SM.

The reason for varing number of warps reside in a SM and number of streaming processors (SP) in SM is justified, CUDA uses these extra warps available to hide long latency operations such as global memory accesses. When a instruction executed by a thread in a warp waits for a long latency operation initiated by previously initiated instruction, that warp is not selected for execution. If more than one warp is ready to start execution, CUDA runtime uses some priority mechanism. This technique to get work done for other threads when executing one are waiting for long latency operations is known as latency hiding.

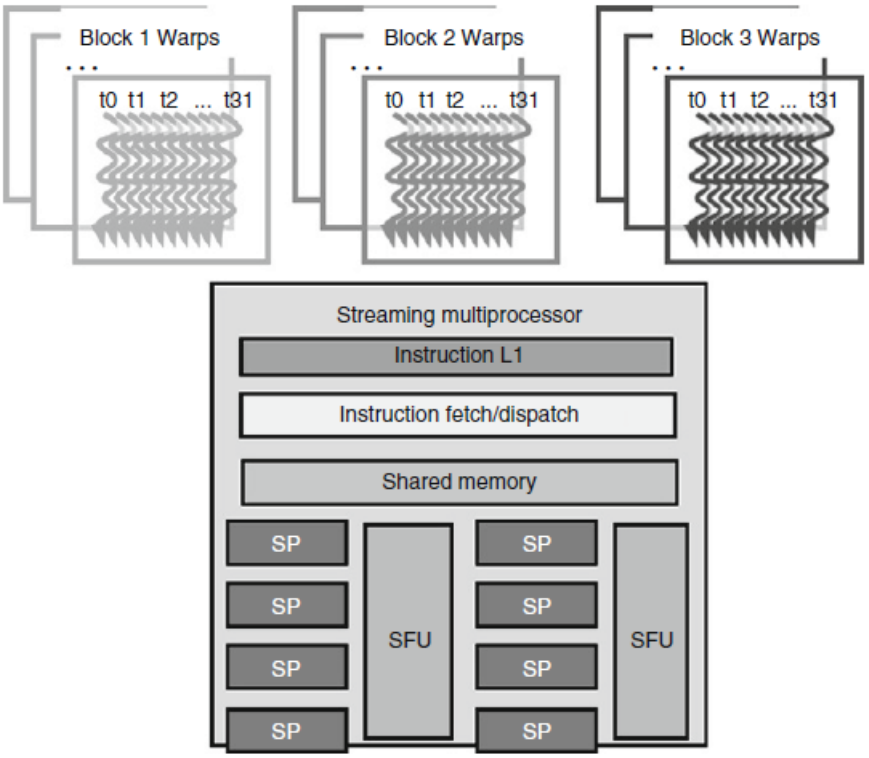

Figure 4.7: Division of block into warps [1]

Warp scheduling is not limited to hide global memory access latencies but it also does it for pipelined floating point arithmetic and branch instructions. With enough warps available, it possible to fully utilize the computing hardware by availing it with warp at any point of time. Switching between warp does not make processing units idle, which is referred to as zero-overhead thread scheduling. This feature of, making long latency operations hidden by executing instructions from other warps to keep processing units busy, makes GPU chips with large portion of GPUs dedicated for floating point execution resources and not cache memories and branch prediction unit as CPU does, possible.

4.3 CUDA Memories

From previous chapters, it is clear that we can achieve parallelism using massive amount of threads. Data to be processed by these threads need to be transferred from CPU's memory to GPUs global memory. Threads access their respective data from global memory with the help of thread identifiers and block identifiers. Although we have created number of threads to achieve parallelism, but this is small fraction of actual capability of GPU hardware. This degrade in performance is due to the global memory which is dynamic random access memory (DRAM). Global memory accesses are time consuming

(hundreds of clock cycles) and has finite memory bandwidth [14]. These long latency memory operations can be tolerated by executing some other threads while assigned are waiting for global memory accesses. After certain limit, this will lead to traffic congestion in global memory access paths, causing no progress of any thread leaving streaming processors idle (SM). To avoid such situations, CUDA provides number of techniques those minimizes the accesses to global memory.

4.3.1 Importance of memory access efficiency

Non-efficient Memory access could be major factor reducing performance of an application. Consider subsequent sample code is in a kernel, the `for` loop in the code is most important with respect to time consumed by it. In every iteration, it accesses a global memory twice and performs one floating point multiplication and one floating point addition operation. The ratio of floating point operations to global memory access is 1 to 1. This ratio is called as Compute to Global memory access (CGMA) ratio.

```
__global__ void kernel(float *M, float *N, float *P,int Max)
{
float val;
thread_id = threadIdx.x + blockIdx.x * blockDim.x;

for(int i=0; i< Max; i++)
{
val += M[thread_id+k] * N[thread_id+k];
}
P[thread_id] = val;

}
```

CGMA ratio is a important factor for performance of CUDA kernel. For e.g. NVIDIA G80 supports 86.4 gigabytes per second (GB/s) of global memory access bandwidth. Maximum number of floating point operations that can be performed on hardware is limited by rate at which input data is loaded from memory. With 4-bytes as a calculation unit in single precision calculations, it can fetch maximum 21.6 (86.4/4) giga single precision data per second. If this is the rate at which data is fed to the system and with CGMA ratio of 1, matrix multiplication kernel will able to execute no more than 21.6 floating point operations second (gigaflops), as each single floating point operation require 4 byte data as a unit. Although this is a good value but it is much less

than theoretical peak performance of G80. Therefore various memories are provided in CUDA architecture which helps in performance gain.

4.3.2 CUDA Device Memory Types

CUDA supports several types of memory that can be used by programmers to achieve high CGMA ratio and high execution speeds of CUDA kernels [1]. Fig. 4.8 shows different memories supported by CUDA system. At bottom there are global memory and constant memory. Host system can access these memories for reading and writing purpose with the help of application programming interface (API) functions. Global memory is explained in previous section. Constant memory is short latency, high bandwidth memory and device has only read only access to these memories [14].

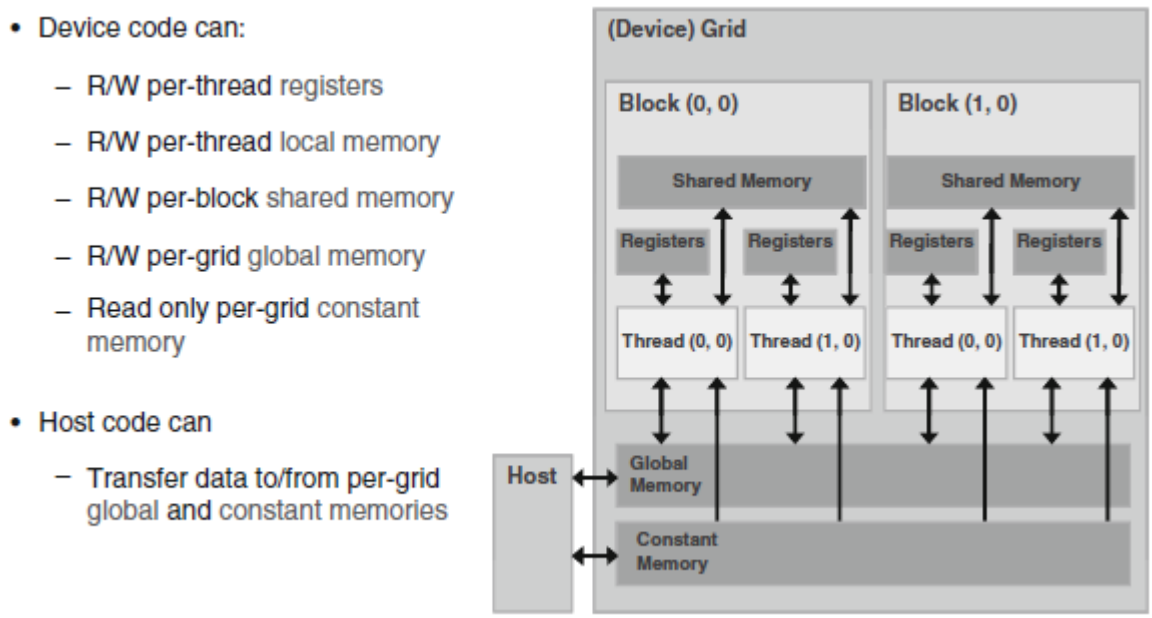

Figure 4.8: CUDA Memories [1]

Registers and shared memory in fig 4.8 are on chip memories. Variables those are in this memory are accessible in parallel manner. Register are allocated for each thread; each thread has different copy of register variable. Kernels use registers to hold the variables those are frequently accessed by a thread and are private part of thread. Shared memory is common to a block of threads; all threads in a block can access shared memory which is common to all the threads. Shared memory is an efficient way of achieving cooperation among the threads and share the intermediate results. By declaring a CUDA variable in one of the types mentioned above, programmer states its visibility and access

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
__shared__ int sharedVar	Shared	Block	Kernel
__device__ int globalVar	Global	Grid	Application
__constant__ int constVar	Constant	Grid	Application

Table 4.2: CUDA Variable type qualifiers

speeds of variable.

Table 4.2 gives the CUDA syntax for the declaring variables into a different memory types. Variable declared with such syntax defines its lifetime and scope. Scope of a variable is range of threads those can access this variable: a single thread, block of thread or complete grid of threads. If scope of a variable is a single thread; a private version of code is created for every thread. e.g. If a kernel declares a variable whose scope is thread, and it is launched with 1 million threads, then 1 million copies of the variable are created for each thread. Each thread initializes and uses its own copy of variable.

Lifetime of a variable specifies portion of a program execution when a variable is available for use: either inside kernel execution or through entire application. If a variable's lifetime is a kernel then it must be defined with the kernel body and it is available to use only for the kernel function. Multiple calls to kernel function then system will not maintain value for variables for previous version. Each invocation must initialize variables before using them. Instead, variables lifetime is application, its value is maintained across kernel function calls. These variables must be declared outside function body. These variables are available to all kernels.

As shown in the Table 4.2, all scalar variables declared in kernel and device functions are placed into registers (Here all non array variables are scalar variables). The scope of these automatic variables are within individual threads. When a kernel function declares an automatic variable, a private copy of variable is generated for every thread that executes a kernel function. When a thread terminates, all of its automatic variables are deleted. Accessing these variables is extremely fast but using it extensively is not feasible because they are limited in number in the hardware.

Automatic array variables are not stored into registers instead they are stored onto global memory. These arrays incurs long access delays and potential access congestions. The scope of these arrays is limited to the thread just like scalar variables. That is, a different copy of variable is created for every thread. Once thread terminated its execution, the contents of these threads also vanishes. It is good practice not to use automatic

array in kernel or device function.

If a variable declaration is preceded by the keyword __shared__, variable is shared variable in CUDA. The same effect can be achieved by adding optional __device__ in front of __shared__ in the declaration. These variables are declared in kernel or device function. The scope of a thread variable is a thread block. A different copy of shared variable is created for a different thread block. The lifetime of a shared variable is kernel function. When a kernel terminates its execution, contents of shared variable are also deleted. Accessing shared variable is extremely fast and highly parallel. CUDA programmers use shared memory to hold a portion global memory that is heavily used in execution phase of kernel. While using shared memory, there is need to change the access patterns of algorithm that focuses on smaller portion of global memory.

If a variable declaration is preceded by __constant__ then variable declared is of type constant memory. The same effect can be achieved by adding optional __device__ in front of __constant__ stating it as a device variable. Declaration of a constant variable must lie outside any kernel or device function. The scope of a constant variable is all grids, only one copy of this variable is created for all threads. All threads sees same copy of constant variable. This value persists through multiple kernel function calls, in fact constant variable is created only once. The lifetime of a variable is entire application. Constant variables are generally used to feed input values to kernel functions. Constant variables are stored onto global memory but are cached for faster access. Constant memory is extremely fast, if certain access patterns are followed. Total size of constant variables in an application is 65,535 bytes. There is need to break data to fit this limitations.

Variables preceded only by __device__ are global variable and are stored in memory. Global memory accesses are slower but variables stored in global memory accessible to all the threads of all kernels. Lifetime of these variable is also an entire application. Global memory can be used to collaborate threads across the blocks, but there is no way to synchronize the data accesses across the thread blocks to ensure data consistency. The only way to get synchronized data is by terminating kernel call. Therefore, global variables are often used to pass information from one kernel invocation to other invocation.

4.3.3 Memory as a limiting factor to parallelism

Although CUDA registers, shared memory and constant memory are very useful in reducing accesses to global memory. It is necessary to take care that, these memory use should not exceed its size. CUDA devices has limited amount of such memory, which limits number of threads that can reside in a streaming multiprocessor for an applica-

tion. Number of threads reside in memory is inversely proportional to usage of these kind of memories.

In the G80 GPU, each SM has 8KB register file (8192 registers), so in total 128K (131,072) for entire processor. Though this number is very large but this allows only small number of registers per thread. G80 can accommodate 768 threads in a SM. In order to fill this capacity, each thread can use only 8k/768 = 10 registers per thread. If number of registers used by each thread is 11 then number of thread that will reside in an SM will reduce. Such a reduction is done at the block granularity as whole block is assigned to a SM not a part of it. For e.g. if block contains 256 threads then 256 threads will be reduced at a time, so total number of threads scheduled on SM are 512, 2/3 of capacity. This will reduce total number of warps available for scheduling, thus processors ability to find some useful work at the time of long latency operations.

Shared memory can also reduce the number of threads assigned to each SM in similar way. In G80, 16 KB of shared memory available in each SM. Shared memory is used by block of threads and G80 can accommodate 8 blocks per SM. In order to properly utilize the shared memory and processing capability, each block may not use 2KB of shared memory. If each block uses more than 2KB of shared memory, then number of blocks that reside in SM should not exceed their limit of using shared memory i.e. 16 KB. If each block uses 5 KB of shared memory then each SM can hold only 3 blocks. These limits of maximum sizes do change but effects are determined at runtime.

Chapter 5

Non Photo-realistic Rendering

The term Non photo-realistic rendering (NPR) is coined by Georges Winkanbach and David Salesin in early '90s to the techniques that left paradigm of photo-realistic rendering far behind [2]. Scientist were seeing the end of road for photorealism and wondering what is laid ahead. In computer graphics, photo-realistic rendering tries to develop virtual images of simulated 3D environments to look like "the real world". So non photo-realistic rendering (NPR) is any technique that produces images of simulated 3D world in a way other than "realism".

Computer graphics community worked on achieving reality, scientist came up with photorealism. Pixar's rendering architecture "Render Everything You Ever Saw" was defined with more emphasis on render the real world artificially. This world is defined by physics: molecules and light. Figure 1 shows example of image created with computer graphics. The computer graphics world has almost realized photorealism. The task remained for them to do, is render artistic renditions using computer graphics.

Photo-realistic Rendering(NPR) research will play a key role in the scientific understanding of visual art and illustrations [15].

We need large amount of details to represent and animate a realistic scene. When human designer creates a design with his hand, it takes a lot of time as well as the efforts. This can be avoided if similar real world images are available but this requirement is not suitable for all the cases. For e.g. Storytelling, we cannot have real world scenery for this case. Here, alternative is to have team of trained experts and let them model and animate content. But this cannot be affordable to smaller organizations. Whereas, it is relatively simple to create NPR images than photo-realistic ones. This way content creation problem could be solved and Due to all above reasons NPR will continue grow and influence its use over photorealism.

Originally, NPR research dealt mainly with static images resulting in extremely beau-

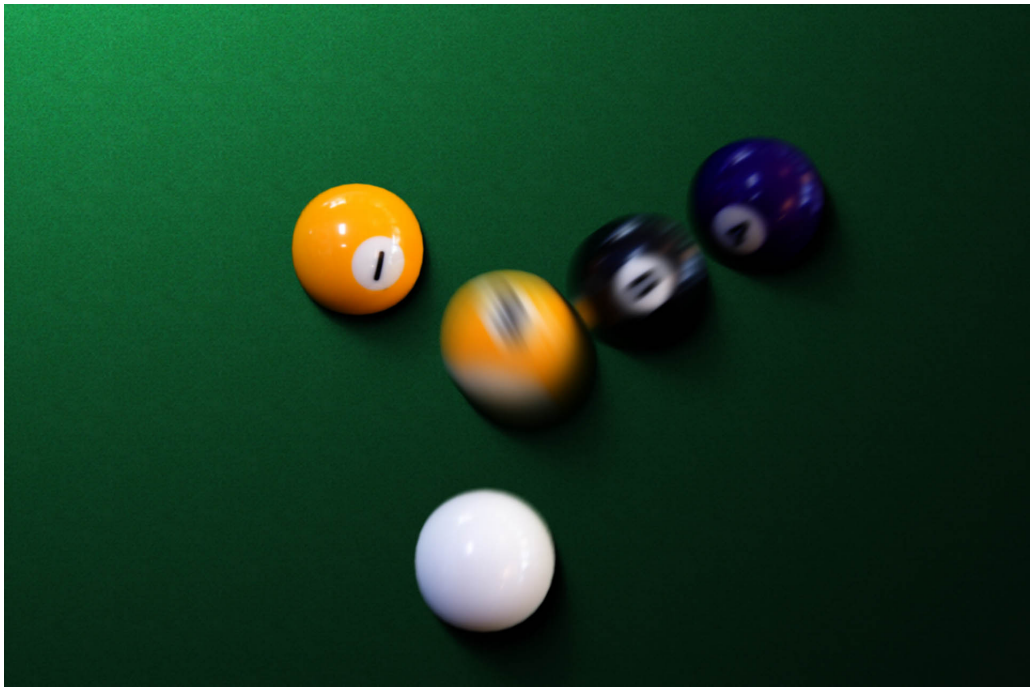

Figure 5.1: Landmark Image in Computer Grphics-Motion Blur, Sources:©1984 Thomas Porter, Pixar [2]

tiful images and can be easily faked for artistic images. Further, researchers produced images that manual rendering could not have achieved. After mastering static images scientist moved towards animation. While depicting primitives defined for model, a special consideration is given to avoid frame to frame flicker. This is termed as *Coherence*.

Whenever we draw a NPR frame, we face following issues related to coherence:

1. *Segmentation* - high level region determination of where to put strokes.
2. *Stroke properties* - number, location ,size, color, orientation and order.
3. *media simulation* - physical simulation of the media on canvas.

To achieve coherence, each of these issues need to be addressed. Media simulation, is least important for coherence, researchers are addressing other two issues.

5.1 Types of NPR

5.1.1 Painterly and Pen-and-Ink Rendering

This type of rendering focuses on producing artistic images from photographic image. These renderings are largely adapted, when these drawn on a white background with

	Photorealism	**NPR**
Approach	Simulation	Stylization
Characteristic	Objective	Subjective
Influences	Simulation of physical processes	Sympathies with artistic processes; perceptual based
Accuracy	Precise	Approximate
Deceptiveness	Can be regarded as "dishonest"	Honest
Level of Detail	Constant level of detail	can adapt level of detail across an image to focus the viewer's attention
Good for representing	Rigid surfaces	Natural and organic phenomena

Table 5.1: Comparison of photorealism and non-photo-realistic rendering(NPR) [16]

black strokes. Shading effect is achieved by increasing density of strokes in that region. Strokes can be used to depict the shape of an object.

5.1.2 Contour Rendering

Contours are the curves that draws the border for objects and depicts shape of an object. Issues in contour rendering are, in very detailed scene large number of contour lines fails to depict shape of an object and sometimes it is computationally infeasible to model all the details in scene that required by an artist [16].

5.1.3 Non Photo-realistic Animation

Non photo-realistic rendering can be applied to non photo-realistic animation; due to frame to frame coherency issues that arises in animation and due to manual work necessary to produce animation sequences. Research work in NPR address such problems of producing coherent image sequences or simplifying user input to generate many frames of rendering.

5.2 Applications of NPR

In many applications, such as, architectural, automotive, industrial, graphics design non photorealism rendering is preferred over photorealism. Non photo-realistic renditions conveys information in better way by omitting extraneous details, by centering on relevant details, by clarifying, simplifying and disambiguating, and showing parts that

are hidden[16]. It provides way to convey information at different levels of details. NPR images are more attractive than photo-realistic images as they posses vitality.

In architectural designs, architecture can present his client with preliminary pencil sketch for house not yet built hiding the details. This way design remains open for modification and client can suggest modifications to design.

In case of images in medical textbooks it is traditional. Here, schematic line drawings depicts much more information than actual real images. Real images gives exact and perfect scene but artistic images emphasize on specific area of image hiding unwanted details[16].

Problems in Non photo-realistic animations are unique due to frame-to-frame coherency issues and amount of manual work necessary to produce animation. Many of NPR technique solves by providing automatic coherency management.

Chapter 6

Algorithm and Implementation

As mentioned in earlier chapters, We have chosen intensity layer based pencil filter[11] algorithm by Yamomato. Yamomato's algorithm extensively uses Line Integral Convolution [17] to generate effect pencil strokes.

Line integral convolution is a vector field visualization technique [17] which uses vector field and white noise to generate an output which gives rough structure of vector space directions. According to [17], Local behavior of vector field can be captured by streamline which starts at center of pixel (x, y) and moves out in positive and negative direction. With this as the base, if LIC is applied to vector field with all vectors in one direction then the streamline captured by LIC algorithm is a straight line. According to subsequent formula given for LIC, output value for the pixel based on streamline. The resulting image looks like pencil strokes drawn in the direction of vector field.

Let F(x,y) be any pixel from white noise image. The forward advection function can be given as

$$P_0 = (x + 0.5, y + 0.5)$$

$$P_i = P_{i-1} + \frac{V(\lfloor P_{i-1} \rfloor)}{\|V(\lfloor P_{i-1} \rfloor)\|} \Delta s_{i-1}$$

The value for function V can be given by following formula.

$V(\lfloor P_{i-1} \rfloor)$ = the vector from the input vector field at lattice point$(\lfloor P_x \rfloor, \lfloor P_y \rfloor)$

To maintain symmetry backward advection is also considered.

$$P_i^{'} = P_{i-1}^{'} + \frac{V(\left\lfloor P_{i-1}^{'} \right\rfloor)}{\left\|V(\left\lfloor P_{i-1}^{'} \right\rfloor)\right\|} \Delta s_{i-1}^{'}$$

Output pixel can be calculated with the following formula using values computed

with forward and backward advection.

$$F^{'}(x,y) = \frac{\sum_{i=0}^{l} F(\lfloor P_i \rfloor)h_i + \sum_{i=0}^{l^{'}} F(\left\lfloor P_i^{'} \right\rfloor)h_i^{'}}{\sum_{i=0}^{l} h_i + \sum_{i=0}^{l^{'}} h_i^{'}}$$

6.1 Intensity Layer Based Pencil Filter Algorithm

LIC algorithm uses vector field and white noise image as input. This algorithm produces white noise for each image that is given as input to LIC algorithm. These are the steps to generate pencil sketch using Enhanced LIC pencil filter [11].

Algorithm

1. Divide the source image into different intensity layers (three in our case). Artist use quantized level of tones instead of continuous tones. To achieve such effect, average intensity of layer is chose as representative intensity of the layer, and convert each layer into binary image making all non white pixels to representative value 6.1 (b,c,d).
2. For all layers except the darkest layer generate the stroke image with the help of Line integral convolution. Different stroke direction is used for each layer to achieve cross hatching effect 6.1(e).
3. Darkest layer is important to represent texture and shape of object, strokes are drawn in the direction of texture of layer if found otherwise it is drawn in user defined direction 6.1(f).
4. Extract the outlines using edge detection algorithm 6.1(g).
5. Add all layers together and adjust the resulting image with paper model to obtain final pencil drawing 6.1(h).

6.2 Implementation

We have implemented two versions of code for pencil sketch algorithm, serial and parallel version. Serial version is developed to compare the performance of parallel versions efficiency. We have used Visual C/C++ as the programming language to develop algorithm. We also used FreeImage library to handle some basic image processing related tasks. Serial version is implemented on Intel©Core i5 processor with 4 GB of DDR RAM.

Parallel version of code is implemented with CUDA C with GPU hardware as GeForce GTX 280. GTX 280 has in all 240 processing cores. The detailed specification for the GPU are listed in table 6.2. CPU used for execution in association with

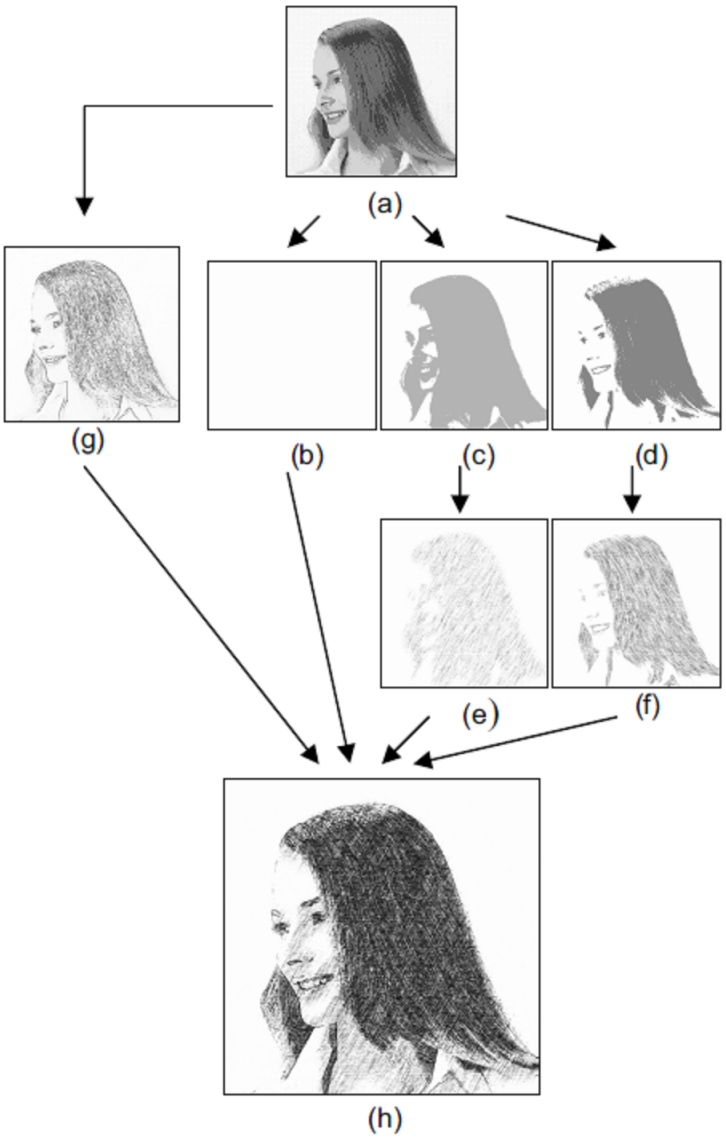

Figure 6.1: Pencil Sketch Algorithmic steps [11]

GPU, is assembled with AMD Opteron processor and 4 GB of DDR memory.

In algorithm, line integral convolution and finding out edge is compute intensive tasks. The remaining tasks can be easily ported to CUDA C and achieve good performance gain with GPU computing power. To get outlines in image, we divided image into small parts, assign it to different multiprocessors for computation and stores parts into shared memories. Shared memories are faster than global memory, its use largely reduces waiting time for memory access. Line integral convolution is used extensively in this algorithm. This algorithm uses LIC with all vectors in vector field in only one direction. This feature left us with option to reduce global memory access for each pixel

Specification	**Value**
No. Thread Processing Clusters(TPC)	10
No. Multiprocessors per TPC	3
No. Streaming Processors per Multiprocessor	8
Total Processing Cores	240
Registers per multiprocessor	16 KB
Share Memory per multiprocessor	16 KB
Constant Cache	64 KB
Global Memory	1 GB
GPU Memory Bandwidth	141 GB/s
Peak Theoretical Performance	1 TeraFlops

Table 6.1: GeForce GTX 280 specification

around half of width size times of image. We have used shared memory to store vector values. As all vectors are in one direction, we can share vector value for remaining pixels. Rest of the global memory accesses exhibit access pattern which can be coalesced and can utilize high global memory bandwidth.

We have used curand library to generate large amount random numbers which we have used for noise image generation and FreeImage library for basic image processing functions. We used visual profiler as tool for profling and Parallel Nsignt debugger.

Chapter 7

Experiments and Results

We have tested our algorithm on different images. Image size we have considered ranges between 512 * 512 and 4304 * 3221. We achieved performance gain of minimum 17x and maximum of 109x. The details of results are listed in table 7. A comparative performance graph of results is as shown in fig. 7.1. From the graph, it is clear that performance increases up to certain limit and stabilizes or falls after reaching certain point. This is because, large number of threads simultaneously running improves performance by providing way to providing work to processors when they are busy in waiting for high latency memory accesses. However, this results in over use of shared memory and interconnection resources which decreases speed up achieved as we increase size of input image.

We worked on various size of images. Following images shows output pencil sketches generated with the implemented algorithm. First image is of size 2592 $\times$ 1944. Second and third are of sizes 4288 $\times$ 2848 and 4304 $\times$ 3221 respectively. Images shown here are with reduced pixel size. We have chosen high resolution size images because low resolution size images does not render fine structures in volume.

Image Size	**CPU Time (In Seconds)**	**GPU Time (In Seconds)**	**Speed Up**
512 * 512	4.63	0.26	17x
1024 * 768	15.3	0.33	47x
2291 * 1500	59	0.7	84x
3648 * 2736	197	1.8	109x
4288 * 2848	220.75	2.62	85x
4304 * 3221	307	2.95	103.8x

Table 7.1: CPU Vs GPU Time on different sizes of images

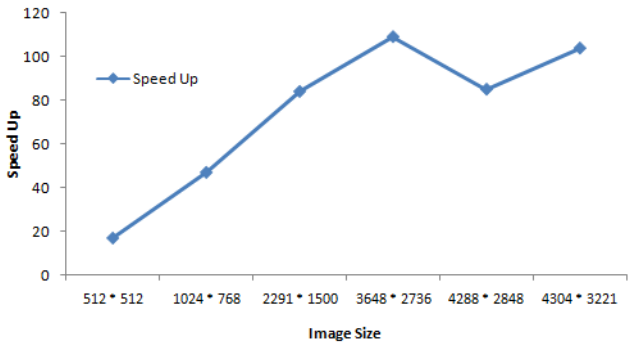

Figure 7.1: Comparison of GPU Vs CPU speed up on different size of images

Figure 7.2: Original image of size: 2592×1944

Figure 7.3: Pencil Sketch for image 7.2

Figure 7.4: Original image of size: 4288 × 2848

Figure 7.5: Pencil Sketch for image 7.4

Figure 7.6: Original image of size: 4304 × 3221

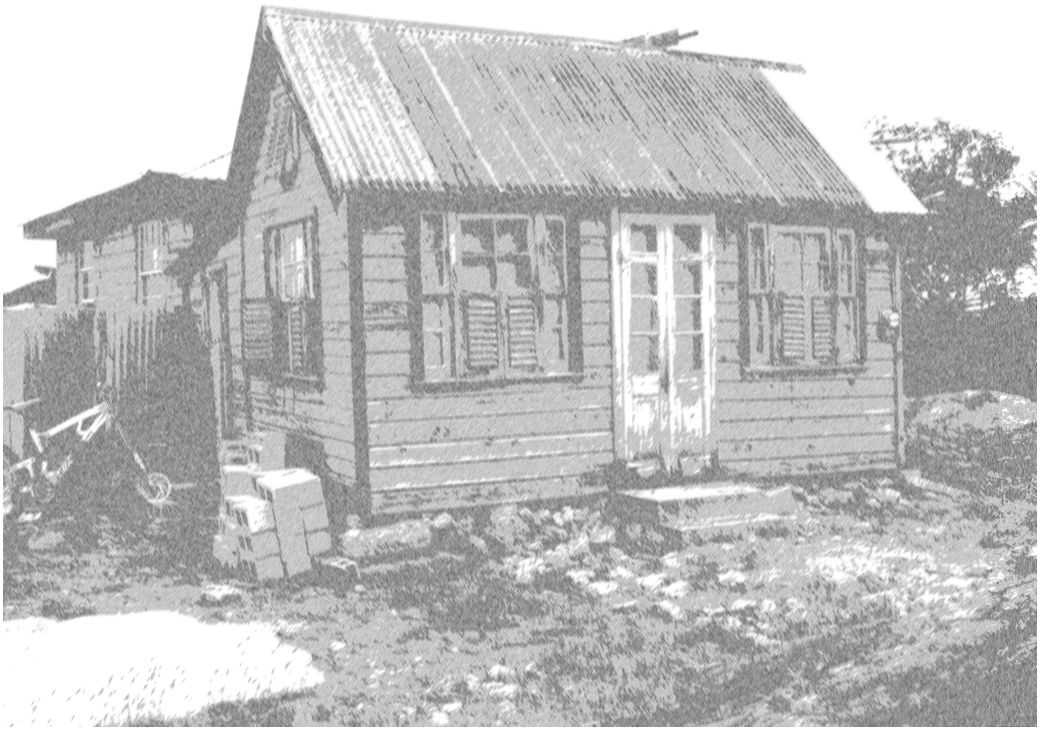

Figure 7.7: Pencil Sketch for image 7.5

Chapter 8

Conclusion And Future Work

8.1 Conclusion

Graphical Processing Units (GPUs) are available with large computing power. We get large number of Giga floating point operations per second (Giga-Flops) for lesser cost than traditional high performance machines and they have large market presence to develop product based on them. CUDA is more preferred programing environment to program graphical processing units(GPUs). Algorithms in non photo-realistic rendering area has tremendous scope for parallelization. There is need for accelerating these algorithms. We have implemented intensity layer based pencil filter algorithm on GPU. We have achieved large performance improvement with CUDA implementation. The performance improvement varies with the input size of image. Large performance gain is achieved with larger size of images due to occupany factor of GPUs.

8.2 Future Work

We have worked on pencil sketch generation using CUDA. This work can be extended for color pencil sketch generation algorithm which is extended work of basic algorithm. Non photo-realistic rendering is an area, which possesses a lot of potential for parallel computing. Similar work can be applied on various algorithms in this area. Applications in this area can reduce the large amount of time for computation. With the advancements in GPUs, it is possible to get much higher performance gain.

Bibliography

[1] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, 2010.

[2] A. Agrawal, "Non-photorealistic rendering:unleashing the artists imagination," 2009.

[3] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, pp. 879 –899, may 2008.

[4] K. Group, "Opencl - the open standard for parallel programming of heterogeneous systems." `http://www.khronos.org/opencl`.

[5] NVIDIA, "Nvidia official website." `http://www.nvidia.com`.

[6] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on cuda," in *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 03*, CSSE '08, (Washington, DC, USA), pp. 198–201, IEEE Computer Society, 2008.

[7] A. Vilanova, "Non-photorealistic rendering." `http://http://www.cg.tuwien.ac.at/courses/CG2/SS2002/NPR.pdf`.

[8] H. Lee, S. Kwon, and S. Lee, "Real-time pencil rendering," in *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, NPAR '06, (New York, NY, USA), pp. 37–45, ACM, 2006.

[9] J. Zhou and B. Li, "Automatic generation of pencil-sketch like drawings from personal photos," in *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, pp. 1026 –1029, july 2005.

[10] D. en Xie, Y. Zhao, and D. Xu, "An efficient approach for generating pencil filter and its implementation on gpu," in *Computer-Aided Design and Computer Graphics, 2007 10th IEEE International Conference on*, pp. 185 –190, oct. 2007.

[11] S. Yamamoto, X. Mo, and A. Imamiya, "Enhanced lic pencil filter," in *Computer Graphics, Imaging and Visualization, 2004. CGIV 2004. Proceedings. International Conference on*, pp. 251 – 256, july 2004.

[12] NVIDIA, "Cuda c programming guide." http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[13] J. Nickolls and W. Dally, "The gpu computing era," *Micro, IEEE*, vol. 30, pp. 56 –69, march-april 2010.

[14] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[15] A. Hertzmann, "Non-photorealistic rendering and the science of art," in *Proceedings of the 8th international symposium on Non-photorealistic animation and rendering*, NPAR '10, 2010.

[16] M. S. S. Irene Liew Suet Fun and A. Bade, "Non-photorealistic outdoor scene rendering: Techniques and application," in *Proceedings of the International Conference on Computer Graphics, Imaging and Visualization (CGIV04)*, 2004.

[17] B. Cabral and L. C. Leedom, "Imaging vector fields using line integral convolution," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, (New York, NY, USA), pp. 263–270, ACM, 1993.

[18] R. Chatterjee, S. Roy, and P. Bhowmick, "A simulation of realistic sketching by randomized pencil strokes," in *Students' Technology Symposium (TechSym), 2011 IEEE*, pp. 29 –34, jan. 2011.

[19] E. B. Lum and K.-L. Ma, "Hardware-accelerated parallel non-photorealistic volume rendering,"

[20] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, pp. 7–17, 2011.

[21] H. Nguyen, *GPU Gems 3*. Addison-Wesley Professional, 2007.

[22] NVIDIA, "Cuda c best practices guide." http://http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf.

[23] M. Harris, "General-purpose computation using graphics hardware." http://www.gpgpu.org.

[24] H. Drolon and F. van den Berg, "Curand library." http://freeimage.sourceforge.net/documentation.html.

[25] NVIDIA, "Cuda reference manaul." http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_Toolkit_Reference_Manual.pdf.

[26] D. Phillips, *Image processing in C*. second ed., 2000.