INDIAN STATISTICAL INSTITUTE

Certificate of Approval

This is to certify that thesis entitled “A study on CUDA architecture and implementation of parallel SAT solver” by Ashis Kumar Dash towards partial fulfillment for the degree of M.Tech in Computer Science at Indian Statistical Institute, Kolkata, embodies the work done under my supervision.

(Prof. Nabanita Das)
ACMU, Indian Statistical Institute
Kolkata

M. Tech. (Computer Science) Dissertation Series

A STUDY ON NVIDIA CUDA ARCHITECTURE AND IMPLEMENTATION OF PARALLEL SAT SOLVER

a dissertation submitted in the partial fulfillment of the requirements for the M. Tech. (Computer Science) degree of the Indian Statistical Institute

By

Ashis Kumar Dash

Roll No. CS0908

Under the Supervision of

Professor Nabanita Das

Advanced Computing and Microelectronics Unit

Indian Statistical Institute

203, Barrackpore Trunk Road

Kolkata-700108

ACKNOWLEDGMENT

I take this opportunity to thank professor Nabanita Das, Advanced computing and Microelectronics Unit, ISI Kolkata for her valuable guidance and inspiration. Her pleasant and encouraging words have always kept my spirit up. I also thank Dr. Ansuman Banerjee and Prof. Bhargab B Bhattacharya for their valuable suggestions.

Finally I would like to thank all my class mates, Dibakar Saha of ACMU, and my family members for their support and motivation to complete this dissertation.

Ashis Kumar Dash
M.Tech(CS)
Roll No: CS0908
Indian Statistical institute
Kolkata

Contents

Abstract

In spite of the enormous progress in the performance of SAT solvers in recent years, still there is strong demand for highly efficient SAT algorithms to solve harder and larger problems. Though there exists huge scope of parallelism, unfortunately, most modern SAT solvers are sequential. Starting with a concise report on CUDA architecture and programming basics, this dissertation presents an implementation of parallel matrix multiplication algorithm on NVIDIA CUDA GPU. Average performance is evaluated executing it on large number of random matrices, in terms of time of completion and speed-up varying the scale of parallelism. Next, this parallel matrix multiplication algorithm is used to solve the Satisfiability problem. In present work, both complete and incomplete SAT solvers have been considered and parallel algorithms are developed and average performance is evaluated executing the algorithms on NVIDIA CUDA GPU for large number of randomly generated Boolean functions.

Key Word: CUDA GPU, Parallel Programming, Kernel, Speed-up, Satisfiability Problem, SAT solver

Chapter 1

Introduction

Thrust for high performance computing power led the computer industry to switch from single CPU based microprocessor to multi-core and many-core models. Since 2003, a class of many-core processors called graphic processing units (GPUs) developed by NVIDIA have led the race for floating point performance. As of 2008 the ratio of peak floating point calculation throughput between many-core GPUs and multi-core CPUs is about 10. Recently developed many-core GPU Fermi has higher performance than this. So GPUs have been found to be the best platform for massive data parallelism. CUDA architecture is based on the heterogeneous platform comprising both CPU and GPU that offers enormous potential to solve complex harder problems efficiently with high speed-up which is still to be explored fully.

The *satisfiability problem* (SAT) is certainly the most studied problem in computer science since it was the first problem proven to be NP-complete by S Cook in 1971. It is seen as the fundamental of computing theory. Its exponential complexity has been challenging the most talented computer scientists for decades. Nowadays, the satisfiability problem evidences great practical importance in a wide range of disciplines, including hardware verification, artificial intelligence, cryptography and it is especially important in the area of Electronic Design Automation (EDA). There is increasing demand for high performance SAT solving algorithms in industry to solve huge and harder problems that show an exponential explosion of the search space. Though there exists huge scope of parallelism, unfortunately, most modern SAT solvers are sequential and fewer are parallel.

In this dissertation a parallel complete SAT solver based on CUDA architecture is designed exploiting an efficient high performance parallel matrix multiplication algorithm. Its performance is compared with the performance of the complete sequential SAT solver. Finally an incomplete parallel SAT solver is designed based on CUDA, and the performances are compared in terms of time of completion when executed on randomly generated Boolean functions.

In chapter 2, CUDA architecture and CUDA programming basics are discussed in brief. Based on this, an efficient CUDA architecture based parallel Matrix multiplication has been presented in chapter 3. The performance evaluation studies and results are also included. In chapter 4, parallel complete and incomplete SAT solvers are developed based on CUDA architecture. This chapter also includes the performance comparison studies in terms of time of completion. Chapter 5 concludes the report.

Chapter 2

CUDA Architecture and CUDA programming

2.1 CUDA Architecture

Following sections present a brief outline of the architecture of NVIDIA CUDA GPU.

2.1.1 NVIDIA GPU Architecture

A modern GPU is organized into 16 highly threaded streaming multiprocessors (SMs). A pair of SMs form a building block of a GPU. Each SM has 8 streaming processors(SPs). So a GPU consists of 128 SPs. Each SP has a multiply- add (MAD) unit, and an additional multiply unit. All running at 1.35 gigahertz. Newly developed GPU Fermi has 32 SMs. So Fermi consists of 256 SPs.

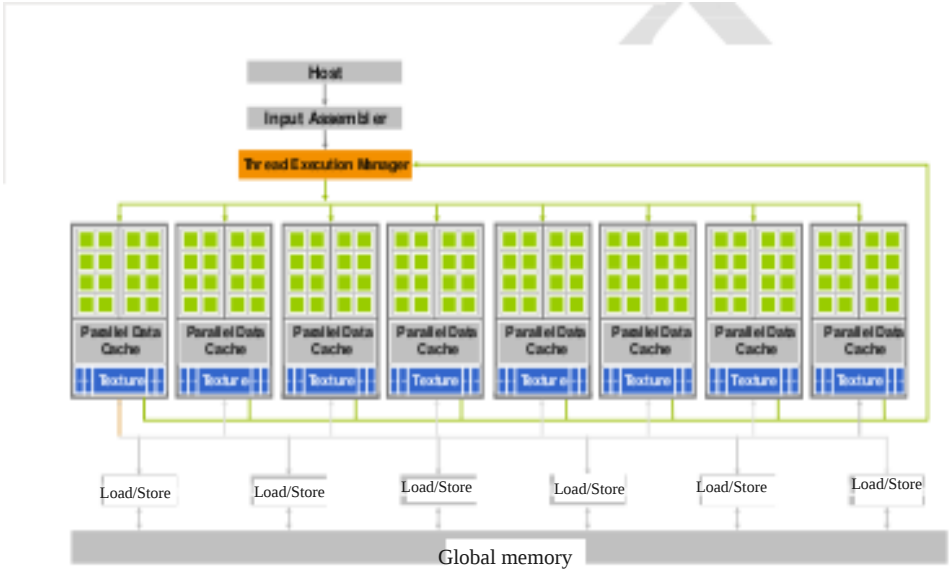


Figure. 2.1: Architecture of a CUDA-capable GPU

2.1.2 Difference between CPU and GPU design

Since 2008 GPUs have led the floating point performance. While performance of micro-processor slowed significantly, the GPUs have continued to improve relentlessly. As per 2008 the ratio of peak floating-point calculation throughput between many-core GPUs and multi-core CPUs is about 10. Recent development has better performance. Such a large performance gap motivated application developers to move their computationally intensive parts of their software to GPU execution. Now question is why there is such a large performance gap between many-core GPU and multi-core CPU. The answer lies in the differences in the fundamental design philosophies between the two types of processors as illustrated in figure 2.2. GPU is designed as numeric computing engine and it will not perform well on some tasks that CPUs are designed to perform well. Most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numeric intensive parts on the GPUs. So CUDA programming model is designed to support joint CPU-GPU execution of an application.

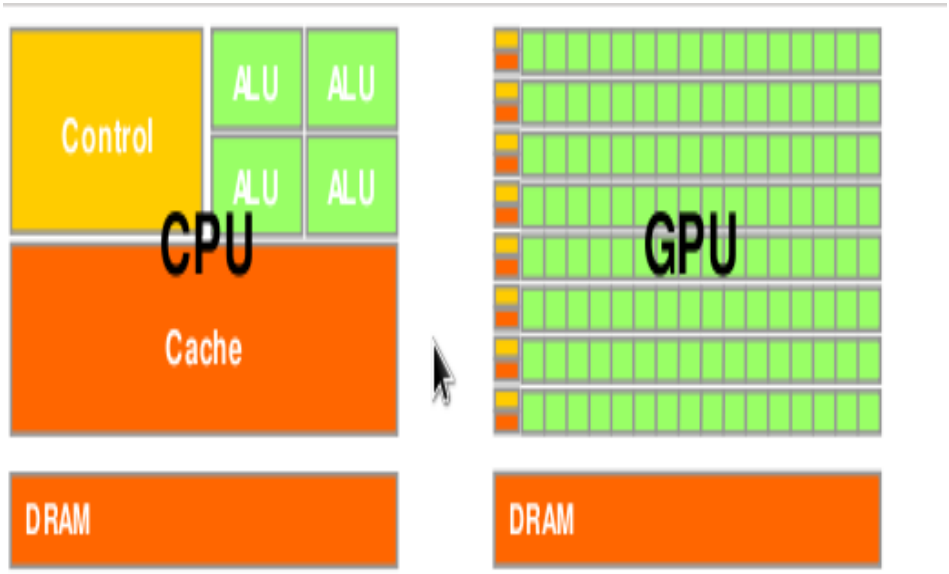


Figure. 2.2: CPUs and GPUs have fundamentally different design philosophies

2.2 CUDA Threads

The fundamental means of parallel execution in CUDA is fine-grained data parallel threads. Launching a CUDA kernel creates a grid of threads. The kernel function specifies the statements that are executed by each individual thread created when the kernel is launched at run-time. Details of kernel function and organization of grid of threads are discussed here.

2.2.1 CUDA Thread organization

Kernel function is a device function which is executed in GPU. Once kernel is invoked it generates grid of threads. All threads execute the same kernel function. These threads have unique co-ordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into a two-level hierarchy using unique coordinates, called blockId and threadId, assigned to them by the CUDA run time system. The blockId and threadId appear as built-in variables that are initialized by the run-time system and can be accessed within the kernel functions. When a thread executes the kernel function, references to the blockId and threadId variables return the appropriate values that form coordinates of the thread.

At the top level of the hierarchy, a grid is organized as a two dimensional array of blocks. The number of blocks in each dimension is specified by the first special parameter given at the kernel launch. The special parameters that specify the number of blocks in each dimension is a struct variable name as gridDim with gridDim.x specifying the number of blocks in the x dimension and gridDim.y specifying the number of blocks in the y dimension. The values of gridDim.x and gridDim.y can be supplied by run-time variables at kernel launch time. Once a kernel is launched, its dimensions cannot change in the current CUDA run-time implementation. All threads in a block share the same blockId values. Figure 2.3 shows a small grid that consists of four blocks organized into a 2X2 array. Each block in the array is labeled with (blockIdx.x, blockIdx.y). For example, Block(1,0) has its blockIdx.x=1 and blockIdx.y=0. The grid was generated by launching the kernel with both gridDim.x and gridDim.y set to 2. At the bottom level of the hierarchy, all blocks of a grid are organized into a three-dimensional array of threads. All blocks in a grid have the same dimensions. Each threadId consists of three components:the x coordinate threadIdx.x, the y coordinate threadIdx.y, and the z coordinate threadIdx.z.

The number of threads in each dimension of a block is specified by the second special parameter given at the kernel launch. Here it refers to the second special parameter as blockDim variable given at the launch of a kernel. The total size of a block is limited to 512 threads, with total flexibility of distributing these elements into the three dimensions as long as the total number of threads does not exceed 512. For example, (512,1,1), (8, 16, 2) and (16,16,2) are all allowable dimensions but (32, 32, 1) is not allowable since the total number of threads would be 1024. Figure 2.3 also illustrates the organization of threads within a block. Since all blocks within a grid have the same dimensions, we only need to show one of them. In this example, each block is organized into 4X2X2 arrays of threads. Figure 2.3 expands block(1,1) by showing this organization of all 16 threads in block(1,1). For example, thread(2,1,0) has its threadIdx.x=2, threadIdx.y = 1, and threadIdx.z=0. Note that in this example, we have 4 blocks of 16 threads each, with a grand total of 64 threads in the grid.

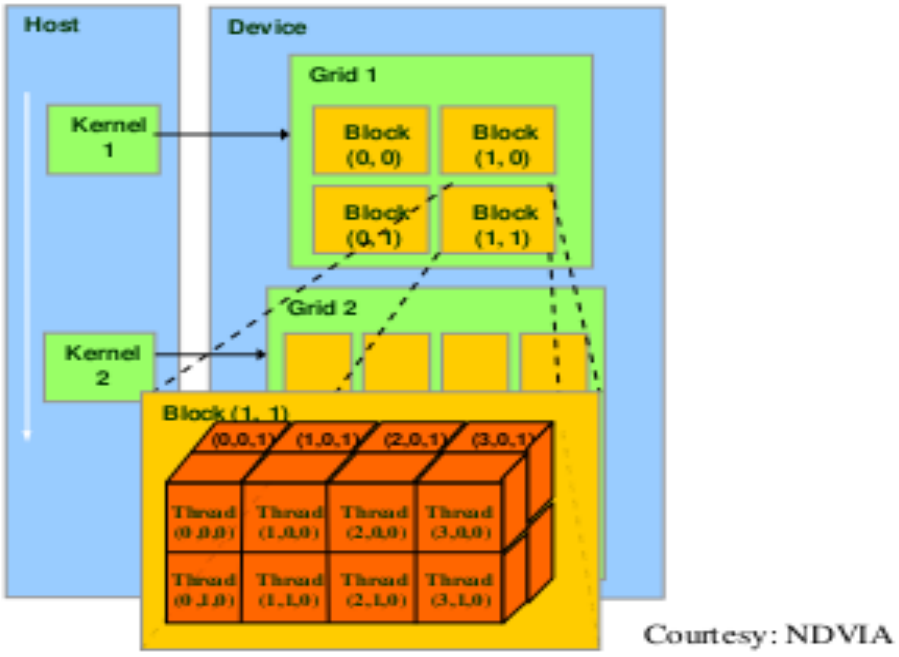


Figure. 2.3: Example of CUDA thread organization

The exact organization of a grid is determined by the special parameters provided during kernel launch. The first special parameter of a kernel launch specifies the dimensions of the grid in terms of number of blocks. The second specifies the dimensions of each block in terms of number of threads. Each such parameter is a dim3 type, which is essentially a struct with three fields. Since grids are 2D array of block dimensions, the third field of the grid dimension parameter is ignored; one should

set it to one for clarity. The thread organization shown in Figure 2.3 is created through a kernel launch of the following form:

```
dim3 dimBlock(4, 2, 2);
dim3 dimGrid(2, 2, 1);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

The first two statements initialize the dimension parameters. The third statement is the actual kernel launch.

2.2.2 Transparent scalability

CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function syncthreads(). When a kernel function calls syncthreads(), all threads in a block will be held at the calling location until everyone else in the block reaches the location. This ensures that all threads in a block have completed a phase of their execution of the kernel before they all move on to the next phase. The ability of synchronizing with each other also imposes execution constraints on threads within a block. These threads should execute in close time proximity with each other to avoid excessively long waiting times. CUDA run-time systems satisfy this constraint by assigning execution resources to all threads in a block as a unit.

This leads to a major trade off in the design of CUDA barrier synchronization. By not allowing threads in different blocks to perform barrier synchronization with each other, CUDA run-time system does not need to deal with any constraint while executing different blocks. This flexibility enables scalable implementations as shown in Figure 2.4. In a low-cost implementation with only few execution resources, one can execute a small number of blocks at the same time, shown as executing two blocks at a time on the left hand side of Figure 2.4. In a high-end implementation with more execution resources, one can execute a large number of blocks at the same time, shown as four blocks at a time on the right hand side of Figure 2.4. The ability to execute the same application code at a wide range of speeds allows one to produce a wide range of implementations according the cost, power, and performance requirements of particular market segments. The ability to execute the same application code at different speeds is referred to as transparent scalability, which reduces the burden on application developers and improves the usability of applications.

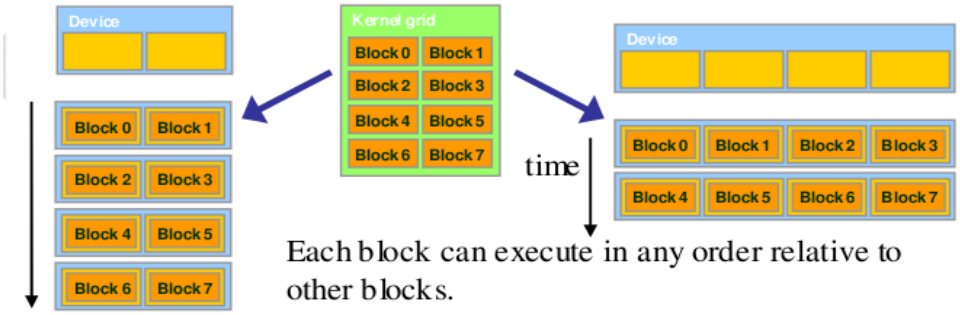


Figure. 2.4: Lack of synchronization across blocks enables transparent scalability of CUDA programs

2.2.3 Thread assignment

CUDA run-time system generates the grid of threads once a kernel is launched. These threads are assigned to execution resources on a block by block basis. In the GeForce-8 series hardware, the execution resources are organized into Streaming Multiprocessors. For example, the GeForce 8800GTX implementation has 16 Streaming Multiprocessors. Up to 8 blocks can be assigned to each SM in the GeForce 8800GTX design as long as there are enough resources to satisfy the needs of all the blocks. In situations where there is an insufficient amount of any one or more types of resources needed for the simultaneous execution of 8 blocks, the CUDA run time automatically reduces the number of blocks assigned to each Streaming Multiprocessor until the resource usage is under the limit. With 16 Streaming Multiprocessors in a GeForc 8800 GTX processor, up to 128 blocks can be simultaneously assigned to Streaming Multiprocessors. Most grids contain much more than 128 blocks. The run-time system maintains a list of blocks that need to execute and assigns new blocks to Streaming Multiprocessors as they complete the execution of blocks previously assigned to them. In the GeForce 8800GTX design, up to 768 threads can be assigned to each SM. This could be in the form of 3 blocks of 256 threads each, 6 blocks of 128 threads each, etc. It should be obvious that 12 blocks of 64 threads each are not a viable option since each SM can only accommodate up to 8 blocks. With 16 SMs in GeForce 8800GTX, there can be up to 12,288 threads simultaneously residing in SMs for execution. So there is a limitation of assigning number of threads to a SM.

2.2.4 Thread scheduling

In GeForce 8800GTX once a block is assigned to a Streaming Multiprocessor, it is further divided into 32-thread units called Warps. The warps is implementation specific and can vary from one implementation to another. Warps are not part of the CUDA language definition. Warps are unit of thread scheduling. Knowledge of warp helps to optimize the performance of CUDA applications. Suppose a block has 256 threads. Then it has 256/32 =8 warps. A SM has maximum 768 threads. That implies up to 24 warps can reside inside a SM at any point of time. For the GeForce-8 series processors, there can be up to 24 warps residing in each Streaming Multiprocessor at any point in time. The SMs are designed such that only one of these warps will be actually executed by the hardware at any point in time. A legitimate question is why we need to have so many warps in an SM considering the fact that it executes only one of them at any point of time. The answer is that this is how these processors efficiently execute long latency operations such as access to the global memory. When an instruction executed by threads in a warp needs to wait for the result of a previously initiated long-latency operation, the warp is placed into a waiting area. One of the other resident warps who are no longer waiting for results is selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution.

2.3 CUDA Memory model

Memory organization is most vital in every architecture. Following sections present how memory is organized in GPU which is significant for reducing access time.

2.3.1 CUDA Device Memory Types

CUDA device has several memories that can be used by programmers to achieve high CGMA ratio and thus high execution speed in their kernels. Figure 2.5 shows these CUDA device memories as implemented in the GeForce 8800 GTX hardware. At the bottom of the picture, we see global memory and constant memory. These are the memories that the host code can write(W) and read(R) calling API functions. The constant memory allows read-only access by the device and provides faster and more parallel data access paths for CUDA kernel execution than the global memory. Above the thread execution boxes in Figure 2.5 are registers and shared memories. Variables that reside in these memories can be accessed at very high speed in a highly parallel manner. Registers are allocated to individual thread; each thread can only access its own registers. A kernel function typically uses

registers to hold frequently accessed variables that are private to each thread. Shared memories are allocated to thread blocks; all threads in a block can access variables in the shared memory locations allocated to the block. Shared memories are efficient means for threads to cooperate by sharing the results of their work.

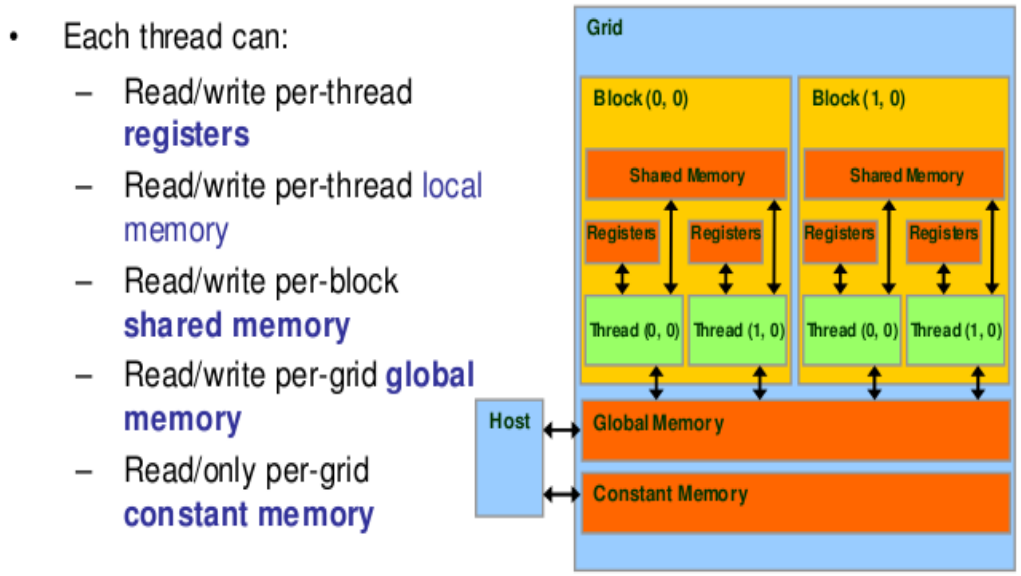


Figure. 2.5: GeForce 8800GTX Implementation of CUDA Memories

2.3.2 CUDA variables

Table 1 shows the CUDA syntax for declaring program variables into the various device memories. Each such declaration also gives its declared CUDA variable a scope and lifetime. Scope identifies the range of threads that can access the variable: by a single thread only, by all threads of a block, or by all threads of the entire grid. If a variable's scope is a single thread, a private version of the variable will be created for each and every thread; every thread can only access its own local version of the variable. For example, if a kernel declares a variable whose scope is a thread and it is launched with one million threads, one million versions of the variable will be created so that each thread initializes and uses its own version of the variable. Lifetime specifies the portion of program execution duration when the variable is available for use: either within a kernel's invocation or throughout the entire application. If a variable's lifetime is within a kernel invocation, it must be declared within the kernel function body and will be available for use only by the kernel's code. If the kernel is invoked several times, the contents of the variable are not maintained across these invocations. Each

invocation must initialize the variable in order to use them. On the other hand, if a variable's lifetime is throughout the entire application, it must be declared outside of any function body. The contents of the variable are maintained throughout the execution of the application and available to all kernels.

Table 1 CUDA variable types

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	register	thread	kernel
Automatic array variables	global	thread	kernel
__device__ __shared__ int SharedVar;	shared	block	kernel
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstVar;	constant	grid	application

All automatic variables except for arrays declared in kernel and device functions are placed into registers. Variables that are not arrays as scalar variables. The scopes of these automatic variables are within individual threads. When a kernel function declares an automatic variable, a private copy of that variable is generated for every thread that executes the kernel function. When a thread terminates, all its automatic variables also cease to exist. Automatic array variables are not stored in registers. Instead, they are stored into the global memory and incur long access delays and potential access congestion. The scopes of these arrays are, same as automatic scalar variable, within individual threads. Once a thread terminates its execution, the contents of its automatic array variables also cease to exist.

If a variable declaration is preceded by keywords "__shared__'' , it declares a shared variable. One can also add an optional "__device__" in front of "__shared__" in the declaration to achieve the same

effect. Such declaration must reside within a kernel function or a device function . The scope of a shared variable is within a thread block, that is, all threads in a block see the same version of a shared variable. A private version of the shared variable is created for and used by each thread block during kernel execution. The lifetime of a shared variable is within the duration of the kernel. When a kernel terminates its execution, the contents of its shared variables cease to exist. Shared variables are an efficient means for threads within a block to collaborate with each other. Accessing to shared memory is extremely fast and highly parallel. If a variable declaration is preceded by keywords "__constant__'', it declares a constant variable in CUDA. One can also add an optional "__device__" in front of "__constant_" to achieve the same effect. Declaration of constant variables must reside outside any function body. The scope of a constant variable is all grids, meaning that all threads in all grids see the same version of a constant variable. The lifetime of a constant variable is the entire application execution. Constant variable are often used for variables that provide input values to kernel functions. Constant variables are stored in the global memory but are cached for efficient access. A variable whose declaration is preceded only by the keyword "__device__" , is a global variable and will be placed in global memory. Accesses to a global variable are very slow. However, global variable are visible to all threads of all kernels. Their contents also persist through the entire execution. Thus , global variables can be used as a means for threads to collaborate across blocks.

2.4 CUDA programming model

CUDA programming aims at data parallelism. In the following sections it has been discussed in brief.

2.4.1 Data parallelism

To write a CUDA program the computing system must consist of a CPU and one or more devices that are massively parallel processors equipped with a large number of arithmetic execution units. In software applications, there are often program sections that exhibit rich amount of data parallelism, a property where many arithmetic operations can be safely performed on program data structures in a simultaneous manner. For example, Let P be the resultant matrix obtained after multiplication of Matrices M, N. Each element of the product matrix P is generated by performing a dot product between a row of input matrix M and a column of input matrix N. The dot product operations for computing different P elements can be simultaneously performed. None of the dot products will effect the results of other.

2.4.2 CUDA program structure

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The phases that exhibit little or no data parallelism are implemented in host code. The phases that exhibit rich amount of data parallelism are implemented in the device code. The program supplies a single source code encompassing both host and device code. The NVIDIA C Compiler (NVCC) separates the two. The host code is straight ANSI C code and is compiled with the host's standard C compilers and runs as an ordinary process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures. The device code is typically further compiled by the NVCC and executed on a GPU device. The execution of a typical CUDA program is illustrated in Figure 2.6. The execution starts with host (CPU) execution. When a kernel function is invoked, the execution is moved to a device (GPU), where a large number of threads are generated to take advantage of abundant data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a grid.

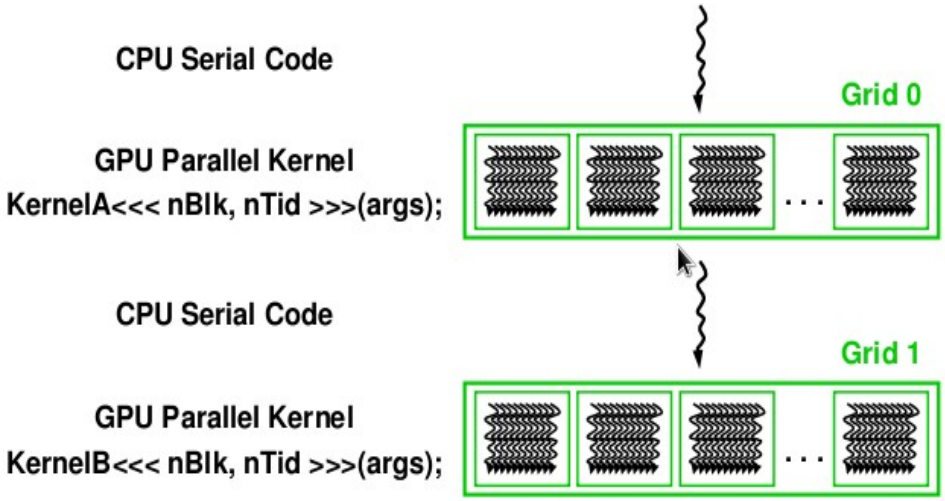


Figure. 2.6: Execution of a CUDA program.

2.4.3 Memory device and data transfer

In CUDA, host and devices have separate memory spaces. Devices are typically hardware cards that come with their own Dynamic Random Access Memory (DRAM). In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer the pertinent data from the host memory to the allocated device memory. Similarly, after device execution, the programmer needs to transfer resultant data from device back to the host and free up the device memory that is no longer needed. The CUDA run time system provides Application Programming Interface (API) function calls to perform these activities. Figure 2.7 shows an overview of the CUDA device memory model which tells about the allocation, movement, and usage of the various memory types available on a device.

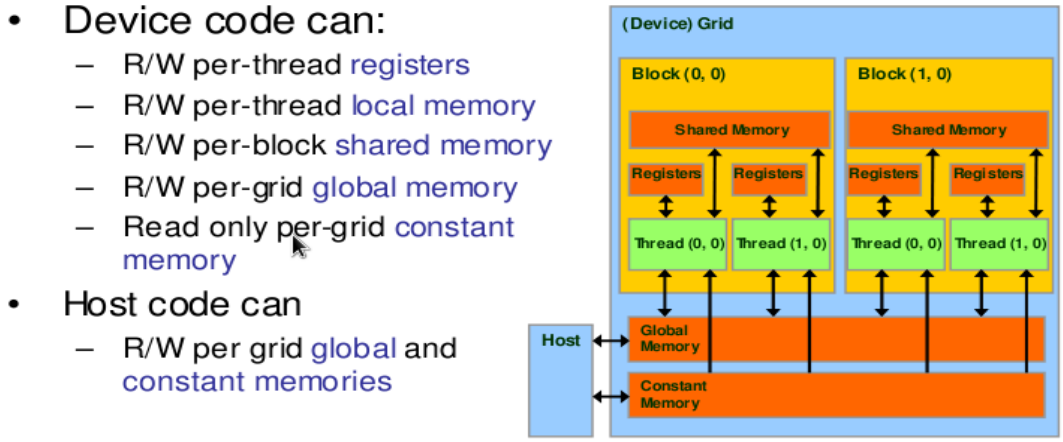


Figure. 2.7: CUDA device memory with a overview of data Transfer

The CUDA memory model is supported by the API function that can be called by CUDA programmers. Two must important API functions are cudaMalloc() and cudaFree(), used for allocating and de-allocating device global memory respectively. Similarly API function cudaMemcpy() is called to transfer data from host to device, device to device and device to host. Detail is given below with their arguments.

```
float *Sd, S;
int   size ;
cudaMalloc((void**)&Sd,size);
cudaMemcpy(Sd,S, size, cudaMemcpyHostToDevice);
cudaMemcpy(S, Sd, size, cudaMemcpyDeviceToHost);
```

During execution of a CUDA program the part that execute data parallelism on device do first transfer the required data from host to device, then invoke the kernel function and finally resultant data is transferred from device to host. Based on above discussion here an algorithm is for matrix multiplication is given in chapter 3 based on CUDA architecture which tries to optimize the data parallelism.

Chapter 3

Parallel Matrix Multiplication on CUDA

3.1 CUDA Architecture based parallel matrix multiplication

In the following sections CUDA based matrix multiplication algorithm is discussed in details.

3.1.1 Skeleton of matrix multiplication

Let M,N and P are three square matrices where M & N are input matrices and P is product matrix. The main steps in host code for matrix multiplication are illustrated below.

```
int main() {
    1. // Allocate and initialize the matrices M, N, P
       // I/O to read the input matrices M and N
          ....
    2. // M * N on the device
       MatrixMulOnDevice(M, N, P, width);
    3. // I/O to write the output matrix P
       // Free matrices M, N, P
          ...
       return 0;
}
```

The main program first allocates the M, N, and P matrices and then performs I/O to read in M and N, in Part 1. Part 2 performs the matrix multiplication. After completing the matrix multiplication in part 3 main function performs the I/O to write the product matrix P and free all the allocated matrices. The part 2 is the main focus. It calls a function, MatrixMulOnDevice() to perform matrix multiplication. The host code calls matrixMulOnDevice(), which is also executed on the host. It is responsible for allocating device memory, performing data transfers, and then activating the kernel that performs the actual matrix multiplication.

3.1.2 Function MatrixMulOnDevice()

Let Md, Nd & Pd are the pointer variables pointing to the first element of a single precision array.

Width is the order of the square matrix. Size here refers to space required for matrix. For simplicity let us assume all matrices are of same order. The code for MatrixMulOnDevice() consists of three parts. The first part allocates device memory for Md, Nd, and Pd, the device counter part of M, N, and P and transfer M to Md and N to Nd. The second part actually invokes the kernel. The third part reads the product from device memory variable Pd to host memory variable P so that the value will be available to main(). It then frees Md, Nd, and Pd from the device memory. Algorithm and the corresponding code are given below.

Algorithm

step-1

Allocate space for M, N, P on device memory

step-2

Load M, N to device memory

step-3

Invoke kernel function

step-4

Read P from device

step-5

Free the allocated space

Code

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
  {
      int size = Width * Width * sizeof(float);
1.    // Load M and N to device memory
      cudaMalloc(Md, size);
      cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
      cudaMalloc(Nd, size);
      cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
      // Allocate P on the device
      cudaMalloc(P d, size);
```

```
2.  // Kernel invocation code
                ...
3.  // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

3.1.3 Kernel Function

In CUDA a kernel function specifies a code to be executed for all threads of a parallel phase. Since all threads of a parallel phase execute the same code, CUDA programming is an instance of the well known Single-Program Multi-Data (SPMD) parallel programming. Let the kernel function for matrix multiplication is MatrixMulKernel(). There is a CUDA specified keyword "__gobal__" in front of the declaration of MatrixMulKernel(). The pseudo code for kernel function using two dimensional thread(threadIdx.x , threadIdx.y) is given below.

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }
    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

when this kernel is launched it creates a single grid of threaded block organized as a 2-dimensional array of threads. Since a thread block can have only up to 512 threads and each thread is to calculate one element of the product matrix, this code can only calculate a product matrix of up to 512 elements. Therefore grid may be organized with multiple blocks. As the above code doesn't use blockId, all the threads implicitly belong to same block. For larger matrix multiplication multiple blocks are needed. Product matrix has to be broken into square tiles. All the pd elements of a tile are computed by block of threads. Let us abbreviate blockIdx.x and blockIdx.y as bx and by, threadIdx.x and threadIdx.y as tx and ty respectively. The corresponding code for kernel function is illustrated below which uses the concept of tile.

```
global__ void Matri xMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = by * TILE_WIDTH + ty;
    // Calculate the column idenx of Pd and N
    int Col = bx * TILE_WIDTH + tx;
    Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row][k] * Nd[k][Col];
    Pd[Row][Col] = Pvalue;
}
```

3.1.4 kernel function using shared memory

The most important part of the kernel in terms of execution time is the for loop that performs inner product calculation. In every iteration of this loop, two global memory accesses are performed for one multiplication and one addition. Thus, the ratio of floating point calculation to global memory access operation is 1 to 1, or 1.0. So use of shared memory is important to reduce execution time. Global memory is large but slow whereas the shared memory is small but fast. A common strategy is to partition the data into subsets called tiles so that each tile fits into the shared memory. The kernel computation on these tiles can be done independently of each other. The concept of tiling can be illustrated with the matrix multiplication example. Figure 2.8 shows a small example of matrix multiplication using multiple blocks. This example assumes that we use four 2X2 blocks to

compute the Pd matrix. Figure 2.8 highlights the computation done by the four threads of block(0,0). These four threads compute $Pd_{0,0}$, $Pd_{1,0}$, $Pd_{0,1}$, and $Pd_{1,1}$.

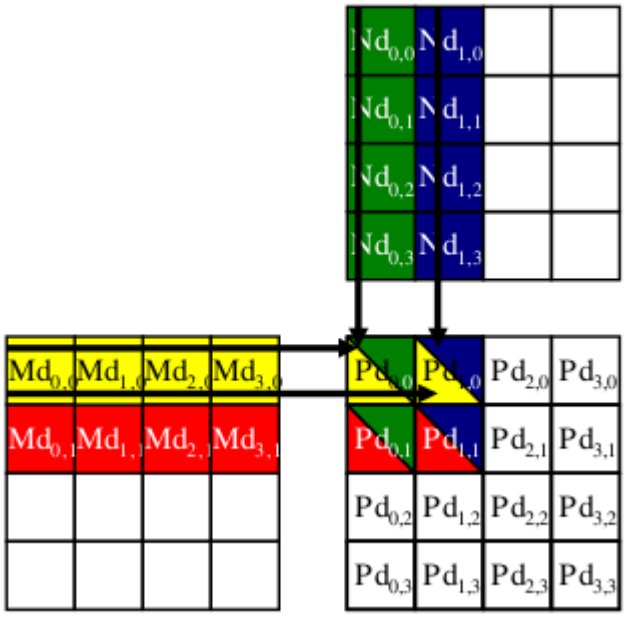


Figure.2.8: A small example of matrix multiplication using multiple blocks

Table 2 shows the global memory accesses done by all threads in $block_{0,0}$. Note that each thread accesses four elements of Md and four elements of Nd during its execution. Among the four threads highlighted, there is a significant overlap of their accesses to Md and Nd. For example, $thread_{0,0}$ and $thread_{1,0}$ both access $Md_{1,0}$ as well as the rest of row zero of Md. In general, we can see that every Md and Nd element are accessed exactly twice during the execution of $block_{0,0}$. Therefore, if we can have all the four threads to collaborate in their accesses to global memory, we can reduce the traffic to the global memory by half.

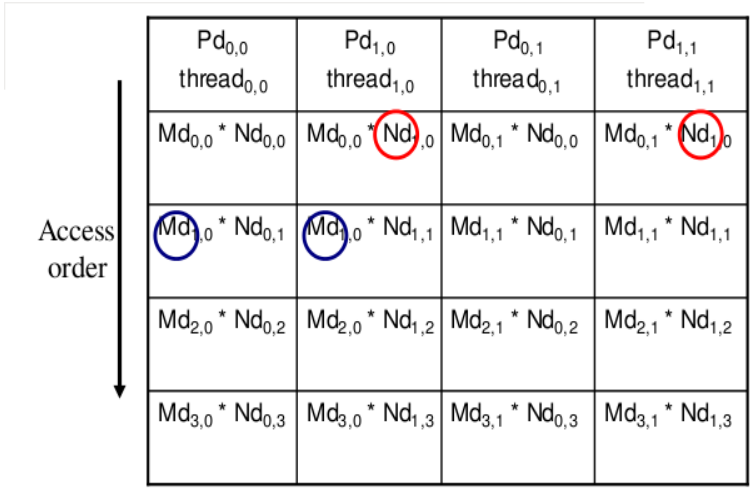


Table 2 Global memory accesses performed by threads in block$_{0,0}$

It is observed that the potential reduction of global memory traffic in matrix multiplication is proportional to the dimension of the blocks used. With NxN blocks, the potential reduction of global memory traffic would be N through collaboration between threads.

Now an algorithm can be designed where threads collaborate to reduce the traffic to the global memory. The basic idea is to have the threads to collaboratively load Md and Nd elements into the shared memory before they individually use these elements in their dot product calculation. Keep mind that the size of the shared memory is quite small and one must be careful not to exceed the capacity of the shared memory when loading these Md and Nd elements into the shared memory. This can be accomplished by dividing the Md and Nd matrices into smaller tiles. The size of these tiles is chosen so that they can fit into the shared memory. In the simplest form, the tile dimensions equal to those of the block. After the two tiles of Md and Nd are loaded into the shared memory, these values are used in the calculation of the dot product. Note that each value in the shared memory is used twice. For example, the Md$_{1,1}$ value, loaded by Thread$_{1,1}$ into Mds$_{1,1}$, is used twice, once by thread$_{0,1}$ and once by thread$_{1,1}$. Mds is a pointer to shared memory. By loading each global memory value into shared memory so that it can be used multiple times, we reduce accesses to the global memory. In this case, the number of accesses to the global memory is reduced by half. The reduction is by a factor of N if the tiles are of NxN elements. Table-3 illustrates this in details.

	Phase 1			Phase 2		
$T_{0,0}$	$\mathbf{Md_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	$\mathbf{Md_{2,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,2}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	$\mathbf{Md_{3,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,2}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,1}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	$\mathbf{Md_{2,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,3}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,1}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{Md_{3,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,3}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Table 3 Execution phases of a tiled matrix multiplication algorithm

So for matrix size 4 and tile size 2 we need two phases to complete each dot product. In general, number of phases = N/Tile_width where N is Matrix dimension. Based on this algorithm the code for kernel function is as follows.

```
global__ void Matri xMulKernel(float* Md, float* Nd, float* Pd, int Width)
 {
    __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__float Nds[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    // Identify the row and column of the Pd element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    int Pvalue = 0;
    // Loop over the Md and Nd tiles required to compute the Pd element
```

```
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of Md and Nd tiles into shared memory
        Mds[tx][ty] = Md[m*TILE_WIDTH + tx][Row];
        Nds[tx][ty] = Nd[Col][m*TILE_WIDTH + ty];
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[tx][k] * Nds[k][ty];
        Pd[Row][Col] = Pvalue;
    }
}
```

Using this algorithm, CUDA based parallel matrix multiplication Code is designed to verify the performance in the following sections and chapters.

3.2 Performance of parallel vs sequential matrix multiplication

In the following sections the performance of parallel matrix multiplication against sequential one has been studied.

3.2.1 System Specification

For this work, the Machine that is used has the following specification. It is a Hp xw8600 workstation. Its core is Intel Xeon E5405, core clock :2000 Mhz, FSB :1333MHz, L2:12MB,Multiplier 6, ,socket LGA771,Data width:64 bit and its family is Harpertown. This machine possesses NVIDIA GPU, Quadro FX 3700. Its core is G92 with core clock :500 MHz, Memory clock :800 MHz, Memory size : 512 MIB, Memory type :256- bit GDDR3, Memory bandwidth :51.2, 112 number of streaming processors, active block capacity 768 and warp :32 threads.

3.2.2 Performance parallel matrix multiplication on basis of block size

The parallel matrix multiplication code, written using the algorithm discussed in the section 3.1 has been executed on the machine as described in subsection 3.2.1. The elements of the matrices are randomly generated floating point numbers of single precision. Here the variable block size has been taken to study the effect of block size. The execution time has been taken for matrix sizes, 1024x1024 and 1012x1012. This time, which is average of 100 readings includes the time for transferring data to the device and performing the matrix multiplication on device. The aim of taking two different matrix size is to study the effect of block sizes those are divisors of matrix sizes. In this case 16 and 22 are

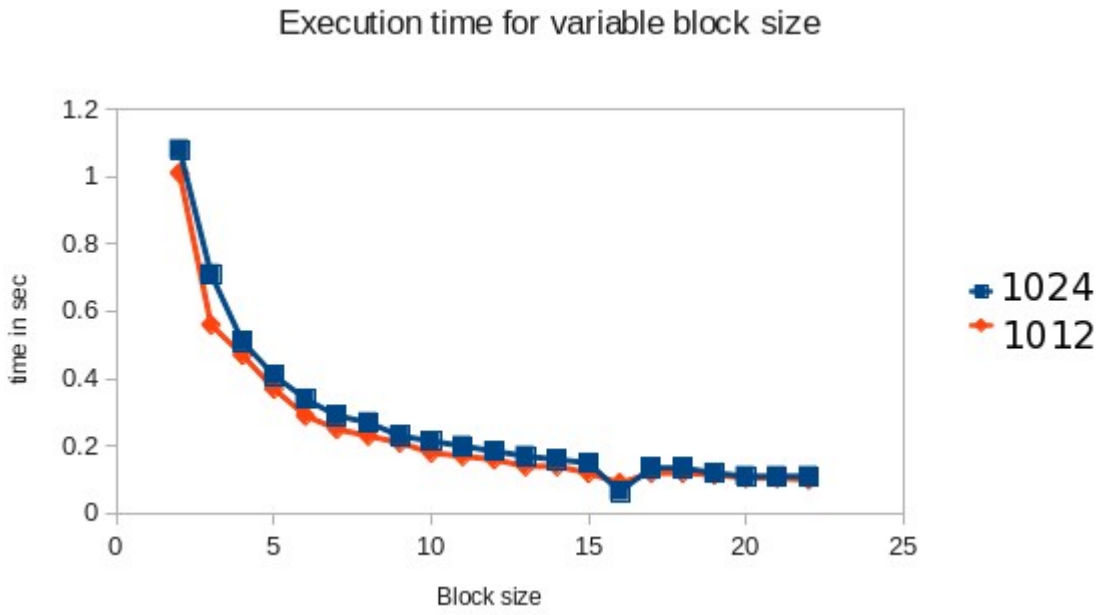
divisor of 1024 and 1012 respectively. Here execution time is considered up to block size 22 (22x22 threads) as the maximum capacity of a block is 512 threads. Table 4 contains details of the execution time.

Table 4

Block size	**Execution time(sec) for Matrix Size (1024X1024)**	**Execution time(sec) for Matrix Size (1012X1012)**
2	1.08	1.01
3	0.71	0.56
4	0.51	0.47
5	0.41	0.37
6	0.34	0.29
7	0.29	0.25
8	0.27	0.23
9	0.23	0.21
10	0.215	0.18
11	0.20	0.17
12	0.185	0.16
13	0.17	0.14
14	0.16	0.14
15	0.15	0.12
16	0.065	0.095
17	0.135	0.12
18	0.135	0.12
19	0.12	0.115
20	0.11	0.105
21	0.11	0.105
22	0.11	0.10

Graph-1 below shows the variation of execution time. First(upper) and second(lower) curve are representing the execution time for matrix size 1024x1024 and 1012x1012 respectively. It decreases as block size increases from 2 to 16. It also decreases as block size increases from 17 to 22, but time for block size 17 to 22 are greater than the execution time of block size 16.

Graph-1



Observation: From graph 1 of the table 4, it is observed that block size 16 results least execution time. It is also observed that the execution time for block size 16 is not only least but also its differences from execution times of block size 15 and 17 are significant. This is because of warp mechanism of CUDA architecture. So with the present system block size 16 is the best choice for matrix multiplication on CUDA as long as the maximum capacity of a block is 512.

3.2.3 Performance of parallel matrix multiplication

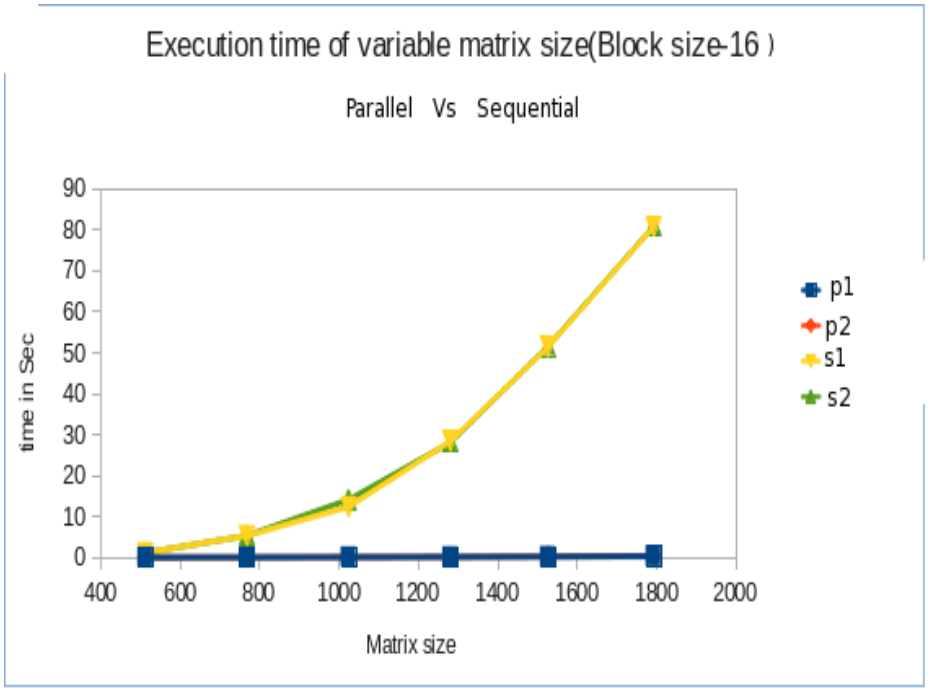
In this section the execution time (parallel vs sequential) is studied for variable matrix sizes. Both parallel as well as sequential code is executed on the same machine as said above. Matrices are taken to be square matrix. The elements of the matrices are randomly generated floating point numbers of single precision. Concluding that block size of 16 has least execution time from section-3.2.2, block size has taken as 16 (16x16 threads) for all succeeding parallel matrix multiplication. Here time taken in sec is average of 100 readings on randomly generated matrices for each size. Table-5 contains the details of execution time and speed up.

Table 5

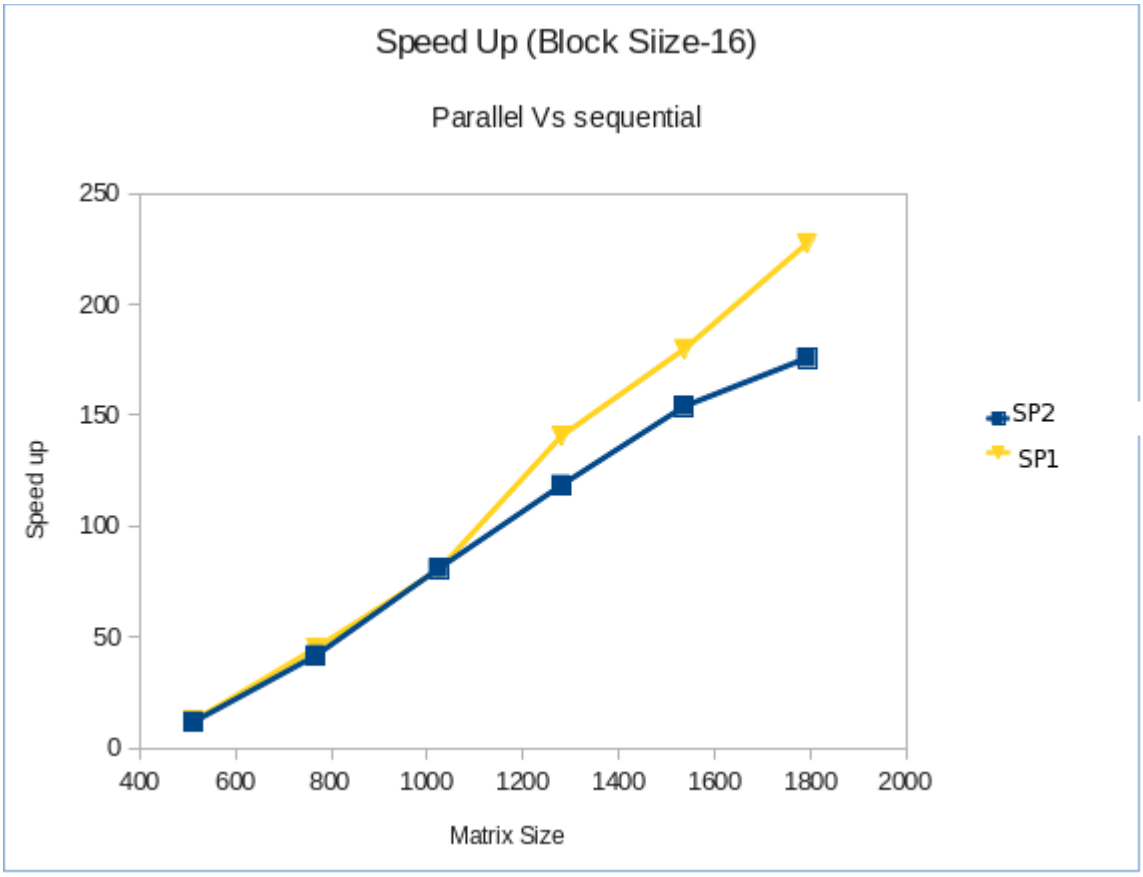
Matrix size	Parallel execution time(sec) P1	Parallel execution time(sec) P2	Sequential execution time(sec) S1	Sequential execution time(sec) S1	Speed up (S1/P1)	Speed up (S2/P2)
512	0.10	0.105	1.215	1.23	12.15	11.71
768	0.12	0.13	5.42	5.42	45.08	41.69
1024	0.155	0.175	12.46	14.17	80.38	80.97
1280	0.20	0.24	28.14	28.45	140.70	118.54
1536	0.26	0.335	51.45	51.63	179.88	154.11
1792	0.355	0.46	80.80	80.90	227.57	175.8

In this table P1 denotes the execution time of parallel matrix multiplication which includes time for transferring data to device, multiplication time on device and transferring the result to CPU. P2 denotes the total execution time of parallel matrix multiplication (execution of main). S1 denotes the execution time of sequential matrix multiplication which includes only the multiplication time. S2 denotes the total execution time of sequential matrix multiplication (execution of main). Graph 2 is representing the data of table 5. Graph 3 is representing speed up.

Graph 2



Graph-3



Here SP1(upper) and SP2(lower) is representing speed up S1/P1 and S2/P2 respectively. It is observed that speed up increases as matrix size increases. This excellent speedup matrix multiplication can be used to solve the computing intensive problems where matrix multiplication can be used,in low time complexity. As the SAT problem is one among them, it has been experimented in chapter 4.

Chapter 4

Parallel SAT solver on CUDA

4.1 Satisfiability Problem

Before presenting the formulation of SAT problem let us define a Boolean function and its CNF representation.

4.1.1 Boolean Function in CNF

A Boolean Formula F is a logical expression defined over variables that takes the value in the set {True, False} which we will identify with {0,1}. A truth assignment to a set V of Boolean variables is a map $\sigma : V \rightarrow \{0, 1\}$. A satisfying assignment for F is a truth assignment σ such that F evaluates to 1 under σ . A Boolean formula has two special form,conjunctive normal form(CNF) and disjunctive normal form(DNF). In this area we are interested for CNF. F is in CNF if it is conjunction(/\) of clauses,where each clause is disjunction(\/) of literals , each literal is either a variable or its negation. For example F= (x\/y)/\(x\/¬y\/¬z)/\z is in CNF with three variables and three clauses. x,y,z,¬y,¬z are literals and x\/y, x\/¬y\/¬z and ¬z are clauses.

4.1.2 Problem Definition

The SAT problem is shorthand for Boolean satisfiability problem. SAT problem refers to the question that given a Boolean expression, determine if there exists an assignment of TRUE(1) or FALSE(0) to all Boolean variables that make the entire Boolean expression to be TRUE. There is another equally important question that there exists no such assignment . Both of them are NP-complete problem [3]. In the first case one assignment is sufficient if such assignment exists. Then we call the Boolean expression is satisfiable, otherwise we call it unsatisfiable which is proven in the second case which needs exhaustive search of all possible assignments. According to the rules of logical equivalence, each Boolean expression formula can be transformed into CNF form which sometimes simplifies the problem to some extent for exposing the underlying structure of the SAT problem, so that a couple of optimization strategies can be applied to reduce the size of the original problem. In addition, Boolean

expressions in CNF can be easily treated as input for SAT solvers. In this dissertation, SAT problem inputs to the solver are assumed to be in in general CNF form.

4.1.3 Earlier Works

The *satisfiability* problem can be solved deterministically in time *poly(n)*.2^n time, where n is the number of literals and *poly(n)* is a polynomial in n. This worst-case upper bound can be decreased to *poly(n)*.c^n, c<2, if we restrict to k-SAT problems, where each clause in the Boolean CNF expression contains at most k literals. In [9], it has been shown that the 2-SAT problem can be solved in polynomial time using a randomized local search procedure. Local search is a well-known heuristic that is applied widely to solve the SAT problem. The best known bound for randomized 3-SAT problem is *poly(n)*.$(4/3)^n$ [10].

Random k-SAT problems exhibit a so-called "Phase Transition Phenomenon" [5], which is when there are exactly k literals in each clause, randomly choose the number of clauses c_k and the number of variables v_k, the probability of the satisfiability of the problem falls sharply from near 1 to near 0 as the ratio $r_k=c_k/v_k$ passes some critical point called threshold. For example, when k=3, the threshold value is about 4.25 (Figure 4.1). However, it is much more complicated to find the threshold value once k is larger than 3. In Figure 4.1, When r_k is close to Y axis, the problem can be easily proved to be satisfiable. Conversely, the problem can be easily proved to be unsatisfiable when it is far from Y axis. The hardest instances appear at the region near the peak (when $r_k \approx 4.25$), during this region, enormous search space needs to be traversed until the solution is found.

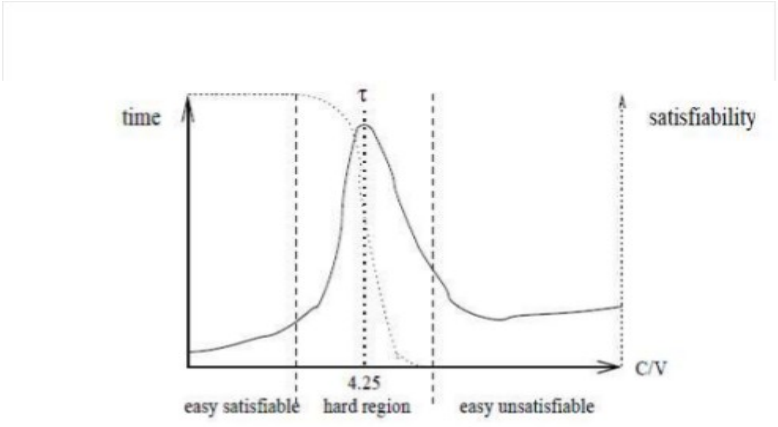


Figure. 4.1: SAT Problem Phase Transition Phenomenon [5]

Random walk strategy [8] for SAT problem is one of the most efficient methods to search the solution for SAT problems by making use of the heuristic variable selection. It evolved from a Pure (unbiased) random walk selection strategy. In this strategy, performance of sequential execution is significantly improved. But it is not suitable for parallel environment. In this case each flip would affect each other unpredictably without guarantee to increase the number of satisfied clauses after operation. GPU4SAT is a parallel incomplete SAT solver [11]. In this SAT solver GPU has been used to solve the SAT problem. Its performance has been compared with that of WalkSAT, an efficient CPU solver. It is a local search method. In this paper it has been proved that its performance is better than that of best WalkSAT. Recently in 2010, NVIDIA CUDA Architecture-based Parallel Incomplete SAT Solver [12] has been developed. In this Solver, the algorithm for parallel SAT solver, based on the cellular genetic algorithm and random walk strategy [8] has been followed.

4.2 SAT Solver

Among all of the SAT solvers, two main categories of SAT solvers are widely studied by researchers. One is Complete SAT solver and another one is incomplete SAT solver.

4.2.1 Complete Solver

Complete SAT solver is the algorithm that checks the satisfiability of the SAT problems. It guarantees to give the result of whether a SAT problem is satisfiable or unsatisfiable. Most of modern complete SAT solvers are based on the classical DPLL algorithm [6]. DPLL itself is still a highly-efficient procedure for SAT problems even under contemporary performance standards. The fundamental principles of DPLL algorithm are backtracking and divide-and-conquer. It firstly simplifies the problem by assigning some values to some variables, so if the rest smaller problem is satisfiable, then the entire formula is satisfiable, otherwise it goes back to assign the opposite values to the appropriate previously assigned variables, and keep doing this recursively until a solution is found or the entire search space is traversed. DPLL actively calls two subroutines *Unit Propagation* and *Pure Literal Elimination* to enhance the efficiency of the algorithm, and recent researches are also eagerly looking for efficient approaches to improve these two functions.

Unit Propagation: Some clauses only contain one literal, there is only one choice for the value of the corresponding variable, then these variables can be safely eliminated from the problem without

affecting the search of the values of other variables. In addition, these eliminations may lead to the deterministic cascades of unit clause which is able to dramatically reduce the size of the original problem to avoid naive search or early detection of assignment conflict which is able to prove the unsatisfiability of the problem.

Pure Literal Elimination: If one variable occurs in the problem with only one form (positive or negative), then all of the clauses that contain this variable can be eliminated from the problem since the Boolean value that makes the corresponding literal true can make all of those clauses be true, and there is only one choice for this value. While *Pure Literal Elimination* is not used in DPLL as intensively as *Unit Propagation*, because finding all of the clauses containing single form of variable is a computation intensive process, and sometimes, it is not worthwhile.

Although modern DPLL-based algorithms have been phenomenally improved, the lack of parallelism make them very complicated to be implemented on large multiprocessors parallel environment [7].

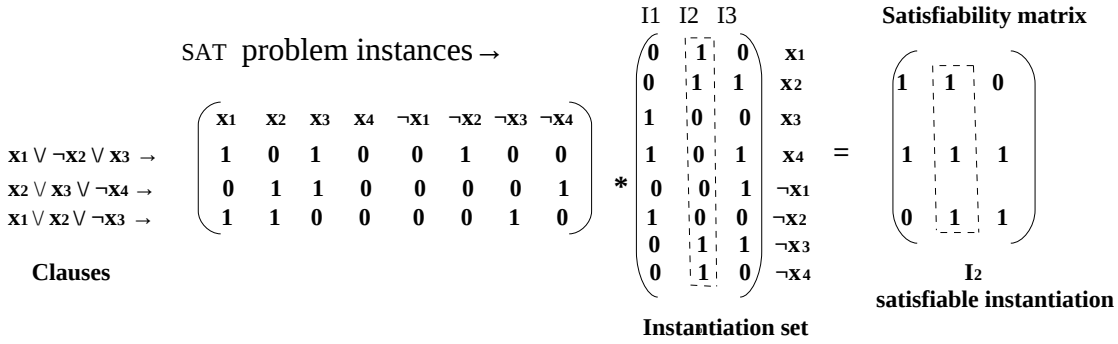
4.2.2 Incomplete Satisfiability Problem Solver

Incomplete SAT solver is the algorithm that finds the solution for the SAT problems during its running time. Incomplete SAT Solvers are not able to prove the unsatisfiability of SAT problem. But its ability of quickly discovering the solution for certain kinds of pretty large satisfiable instances compensates its weakness to a great extent. Incomplete SAT solvers are mostly based on the stochastic local search, and genetic algorithms, most of which are very suitable for parallel computing architecture. This grants the incomplete SAT solvers incomparable advantage of parallelism.

4.3 Matrix representation of CNF Boolean expression

Each SAT problem instance(BF) has a natural matrix representation. In the matrix representation row represents a clause of Boolean expression. If there are n variables present in the Boolean expression, then there will be 2n literals, so 2n columns in the matrix. If a literal presents in the true form, then corresponding value is '1', otherwise '0'. A Boolean expression containing m clauses and n variables can be represented by a binary matrix of size m by 2n. Similarly the matrix of instances is also a binary matrix with 2n rows and maximum 2^n number of columns. We can test the satisfiability of the Boolean expression by multiplying the two matrices and replacing the operation of addition by Boolean 'OR'. If we get at least one column containing only 1's then we say that the Boolean expression is satisfied. The

following example with 4 variables (x_1, x_2, x_3, x_4) and 3 instances (I_1, I_2, I_3) shows that the given function is satisfied for assignment I_2.



4.4 Complete parallel SAT solver Implementation on CUDA

The following sections describe some strategies for parallel complete SAT solver and our implementation.

4.4.1 Performance of parallel SAT solver

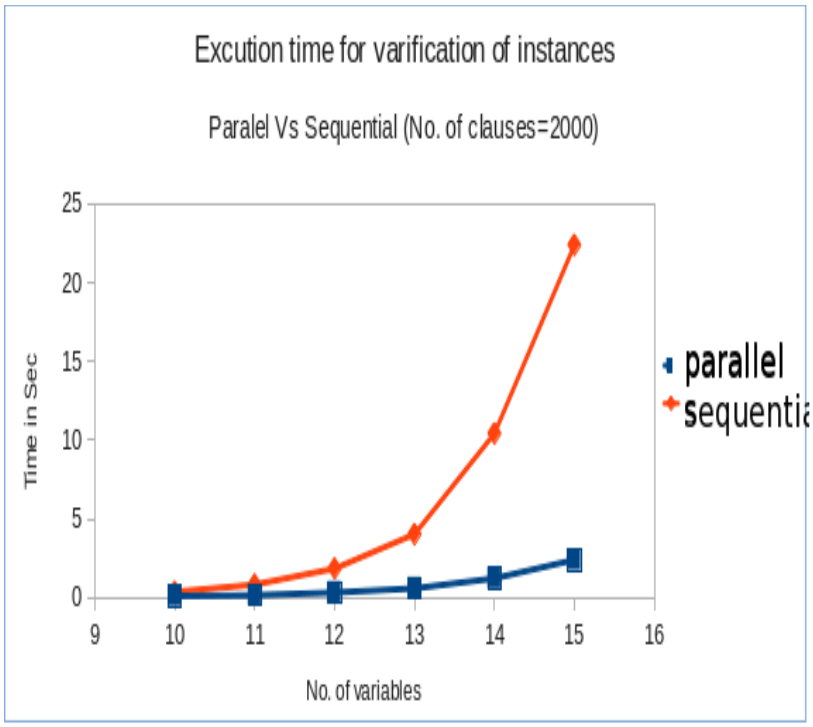
Above discussion tells us that SAT problem can be solved by matrix multiplication. We already concluded that CUDA based parallel matrix multiplication has high performance over sequential matrix multiplication. By generating exhaustive instances and using CUDA based parallel multiplication one can solve the complete satisfiability problem faster than sequential one. Table-6 below gives the detail of execution time in sec. Here the time considered is sum of instances generating time and multiplication time. Program generates all possible instances (worst case). In this case the number of clauses is assumed to be 2000 and block size for matrix multiplication is 16.

Table 6

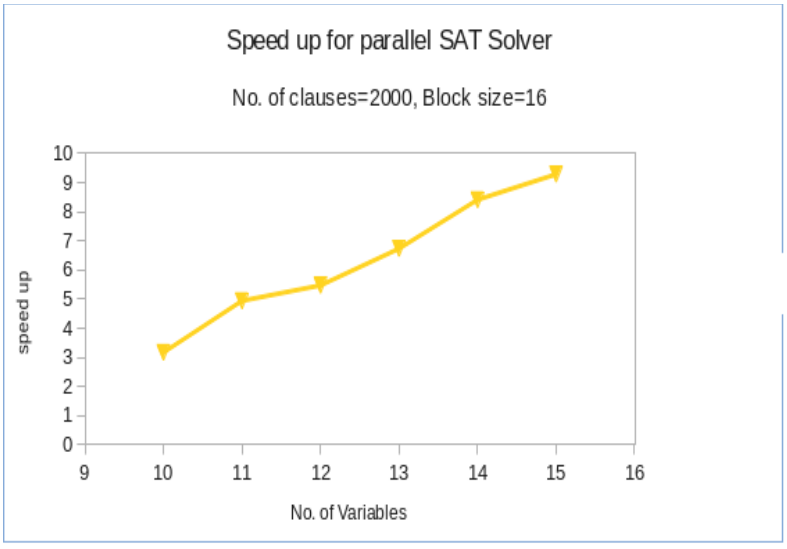
No. of variables	Parallel execution time (in sec)	Sequential execution time(in sec)	Speed up
10	0.12	0.38	3.17
11	0.17	0.84	4.94
12	0.34	1.36	5.47
13	0.60	4.04	6.73
14	1.24	10.43	8.41
15	2.41	22.38	9.28

Graph 4 shows the execution time comparison between parallel(lower) and sequential(upper) SAT Solver. Graph -5 shows the speed up.

Graph 4



Graph 5



Here the performance comparison is done for worst case as the instance matrix contains all possible instances. In this case not only satisfiability is tested but also we can report all the instances that satisfy the Boolean expression.

4.4.2 A better strategy for complete SAT solver

In this discussion a greedy strategy is used in order to verify the satisfiability of SAT problem. At the initial stage a clause generates a set of possible instances that satisfy the 1st clause. In the successive stages each instance considers the next clause in order to generate instances that satisfy present clause as well as all the previous clauses. After the last clause verification all possible satisfiable instances can be reported or problem is unsatisfiable. Details of the strategy is given below.

Suppose the Boolean expression in CNF consists of m number of clauses in n variables. There will be 2n literals. Suppose number of literals in the clauses are not equal. Let n_1, n_2, ..., n_m are number of literals present in clauses C_1, C_2, ..., C_m respectively. Denote a negative variable by 0, a positive variable by 1. From C_1, n_1 distinct instances can be formed where each instance contains exactly one literal (0/1) and rest n-1 places may have any value 0/1 (*don't cares*) such that these instances satisfy the clause C_1. For generating instances let those n-1 places are blank. Now each instance generated in the first stage will generate n_2 instances, assigning some literals in *don't care* positions that satisfy clause C_2. Conflict instances are to be eliminated. So in the second stage there will be at most n_1*n_2 number of instances. Proceeding in this way, at the end the method will generate at most $n_1*n_2*...*n_m$ number of instances. But actual number is less than this due to conflict and repetition. The following example with two clauses in four variables illustrates the method. Let the clauses be $C_1 : x_1 \vee x_3 \vee \neg x_4$ and $C_2: x_2 \vee \neg x_3 \vee \neg x_4$ respectively. The first clause generates following three satisfiable instances where blank space may take any value 0 or 1.

1 _ _ _ , _ _ 1_ , _ _ _ 0.

Now each instance generated will verify second clause to generate new instances those satisfy both the clauses. First instance generates three, second two as there is one conflict (x_3,$\neg x_3$) and third one three (two new & one remains same) instances.

1 1 _ _ , 1 _ 0 _ , 1_ _ 0.

_ 1 1_ , _ _ 1 0.

_ 1 _ 0 , _ _ 0 0 , _ _ _0.

The blank spaces (hyphenated) may take any value 0 or 1. The eight possibilities so generated, in which each of first seven instances generates four and last one eight instances will satisfy both the clauses. We cannot have more than these possibilities that satisfy both the clauses.

If each stage generates the instances in parallel, then we need m stages for a CNF Boolean expression consisting of m clauses. In stage-m we get either all possible satisfiable instances (after assigning all possible values to blank spaces left) or conclude that the Boolean expression is unsatisfiable. At each stage the computation can be done in parallel but the stages are to be executed sequentially. It is a complete solver and produces all assignments of variables which make the function true. However, the scope of parallelism is limited.

4.5 A strategy for Incomplete parallel SAT solver on CUDA

In this strategy the high performance parallel matrix multiplication on CUDA has been used to test the satisfiability of CNF Boolean expression. This strategy is organized into stages. Each stage has two sub-stages. In the first sub-stage a fixed number of instances (keeping in mind the capacity of the GPU device) has been generated and used for testing satisfiability. In the second sub-stage a particular row value of the above instances is made '1', so that those are always satisfying a fixed clause (chosen arbitrarily). The instance matrix so generated has been used to test the satisfiability of the CNF Boolean expression. If it has a satisfiable instance, then the process will be terminated and the satisfiability will be reported. Otherwise, it will go for next iteration. An outline of the proposed algorithm is given below:

Algorithm

1. *Input : Matrix A of the clauses of Boolean Function (BF) in n variables,*
   L: Upper bound on instances,
   I: Fix number of instances in each step for testing

2. *Initialization:*
   {
   Select a Clause C arbitrarily from A
   v = position of a literal=1 in C
   }

3. For j=0 to L/I

{

if j.I ≥ 2^n -1 then report SAT is unsatisfiable and terminate

else if j=⌈L/I⌉ then report SAT is undecidable and terminate

else generates instance matrix B (2n X I), using the binary representation of decimal number from j.I to (j+1).I-1.

multiply A & B and verify the satisfiability

if True, report SAT is satisfiable and terminate

else modify B by replacing all the v-th row elements by '1'

multiply it with A and check the satisfiability

if True, report SAT is satisfiable and terminate

else next j

}

The code, written using above algorithm has been executed in the same Machine as said above and average execution time (in sec) over 100 readings has been taken for a given number of variables. The number of clauses is fixed to be 2000 for all cases. This is our proposed algorithm for parallel incomplete solver. The Table 7 below shows the execution times for parallel incomplete, parallel complete (which terminates once the function is satisfied or the instance set is exhausted), parallel complete (worst case), and sequential (worst case), which always tests all the instances exhaustively, SAT solvers.

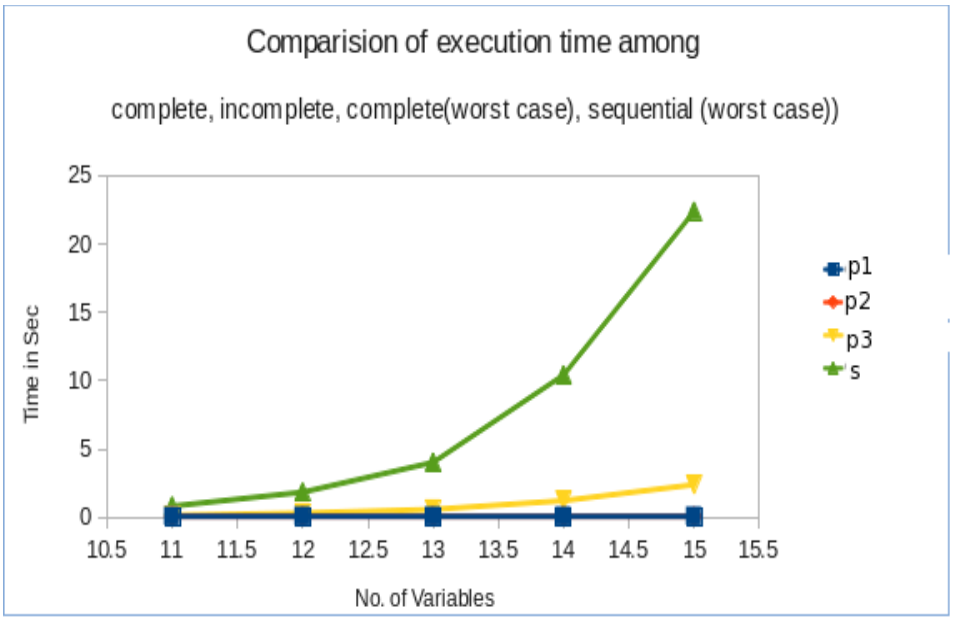
Table 7

No. of variables	Incomplete execution time in sec(2000 clauses)-P1	Complete execution time in sec(2000 clauses)-P2	worst case Complete execution time in sec(2000 clauses)-P3	Sequential execution time in sec(2000 clauses)-S
11	0.0875	0.0885	0.17	0.84
12	0.0885	0.089	0.34	1.36
13	0.0885	0.089	0.60	4.04
14	0.089	0.089	1.24	10.43
15	0.089	0.0891	2.41	22.38

The graph 6 below presents the execution time given in the above table 7. P1, P2, P3 and S are representing the execution times of parallel incomplete, parallel complete, parallel complete (worst case) and sequential (worst case) SAT solver respectively. Though it is observed that the complete and incomplete SAT solver has better performance than complete (worst case) and sequential (worst case). As the number of clauses are very large in comparison to the number of variables and the matrix

for BF is generated randomly, the satisfiability is decided in few stages. Instead of random generation, if set of hard test cases can be used, we can differentiate the performance of incomplete from complete SAT solver.

Graph 6



Chapter 5

Conclusion and Future Work

In this work it has been observed that CUDA architecture has a high performance for parallel matrix multiplication over sequential matrix multiplication. Block size 16 is the best choice for matrix multiplication as long as maximum number of threads is 512. Complete SAT solver (worst case) has shown better performance over sequential SAT solver (worst case). CUDA parallel matrix multiplication has shown a high speed up to verify the satisfiability of instances. But no conclusion can be drawn about the performance of incomplete SAT solver against complete SAT solver. Perhaps the limitation lies in random generation of clauses. If particular set of Boolean functions which are hard for sequential SAT solver will be used to test the performance in term of execution time both for complete and incomplete SAT solver, then their performance may be differentiated. It could be concluded better if its time complexity can be compared with that of existing parallel SAT solvers. Performance as shown in this report can be improved if the generation of instances could be parallelized. In this case random generation of clauses may have repetition. Generation without repetition will do better. Last but not the least, the choice of arbitrary clauses in incomplete strategy may be replaced by a better one.

References

[1] NVIDIA CUDA Programming Guide. Version 3
http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_Program

[2] Alan Kaminsky, http://www.cs.rit.edu/ ark/spring2009/736/ 4005-736 Parallel Computing II, GPU Computing: Introduction to GPGPU and CUDA.

[3] Cook, Stephen "The complexity of theorem proving procedures" Proceedings of the Third Annual ACM Symposium on Theory of Computing. pages: 151-158. Year 1971.

[4] Papadimitriou, C.H, Computational Complexity. 1994. Addison–Wesley.

[5] D.Singer."Parallel resolution of the satisfiability problem: a survey." In E.Talbi,editor. Parallel Combinatorial Optimization. John Wiley and Sons, pages: 123-147, Year 2006 .

[6] Davis, Martin, Putnam, Hillary "A Computing Procedure for Quantification Theory" In Journal of the ACM 7. pages: 201-215, Year 1960.

[7] D.Singer, and A.Monnet. "Jack-SAT: A New Parallel Scheme to Solve the Satisfiability Problem (SAT) based on Join-and-Check." In Proceedings 6th. Int. Conf. on Parallel Processing and Applied Mathematics, PPAM 2007, Gdansk, Poland, Springer Verlag LNCS 4967, pages: 249-258, Year 2008.

[8] Wei and Bart Selman "Accelerating Random Walks," In Principles and Practice of Constraint Programming, pages: 61-67, Year 2002 .

[9] C.H. Papadimitrious. “On selecting a satisfying truth assignment”. In the proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science,FOCS'91,pages 163-169,1991.

[10] Christos Papadimitrious, Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch,Ravi Kannan, Jon Kleinberg “A deterministic Algorithm for k-SAT based on local search".

[11] Herv´ Deleau, Christophe Jaillet, and Micha¨l Krajecki “GPU4SAT: solving the SAT problem on GPU ”.

[12]Alan Kaminsky, Stanisław Radziszowski, James Heliotis and Yandong Wang “NVIDIA CUDA Architecture-based Parallel Incomplete SAT Solver”.