Technical Report

simCUDA: A C++ based CUDA Simulation Framework

Abhishek Das and Andreas Gerstlauer

UT-CERC-16-01

May 20, 2016

Computer Engineering Research Center
Department of Electrical & Computer Engineering
The University of Texas at Austin

201 E. 24th St., Stop C8800
Austin, Texas 78712-1234

Telephone: 512-471-8000
Fax: 512-471-8967

http://www.cerc.utexas.edu

The University of Texas at Austin
Electrical and Computer Engineering
Cockrell School of Engineering

Abstract

simCUDA: A C++ based CUDA Simulation Framework

Abhishek Das, M.S.E

The University of Texas at Austin, 2016

Supervisor: Andreas Gerstlauer

The primary objective of this report is to develop a CUDA simulation framework (simCUDA) that effectively maps the existing application written in CUDA to be executed on top of standard multi-core CPU architectures. This is done by specifically annotating the application at the source level itself, and making the relevant changes required for the application to run in a similar and functionally equivalent manner on a multi-core CPU as it would run in a CUDA-supported GPU. The simulation framework has been developed using C++11 threads, which provides an abstraction for a thread of execution, as well as several classes and class templates for mutexes, condition variables, and locks, to be used for their management. As an extension to the simulation framework, the basic block sequence of execution on a per thread basis is also computed for analysis. This information can in turn be used to derive the basic block sequence of execution on a per warp basis, and thus emulate and replicate real-world behavior of a GPU.

Table of Contents

List of Tables v

List of Figures vi

Chapter 1 Introduction 1

- 1.1 simCUDA Overview 2
- 1.2 Design Decisions 3
- 1.3 Related Work 4
- 1.4 Report Outline 5

Chapter 2 CUDA Programming Model 6

- 2.1 Kernels 6
- 2.2 Threads 6
- 2.3 Blocks 7
- 2.4 Grids 9
- 2.5 Warps 9
- 2.6 Memory 11
- 2.7 Programming model 12
- 2.8 Compilation and Binary Generation 14

Chapter 3 Functional Translation of CUDA to C++ 16

- 3.1 Defining CUDA-specific datatypes 16
- 3.2 Passing implicitly defined variables to kernel 17
- 3.3 Launching threads 19
- 3.4 Serialization of CUDA thread blocks 20
- 3.5 Synchronization primitive between threads 22
- 3.6 Modifying CUDA device-specific and host-specific functions 25
  - 3.6.1 __global__ 25
  - 3.6.2 __device__ 25
  - 3.6.3 __host__ 26
- 3.7 Modification of memory types 26

3.7.1 Shared Memory 27
3.7.2 Global Memory 28
3.7.3 Constant Memory 28
3.7.4 Texture Memory 28
3.8 Modifying memory operations 29
3.8.1 cudaMalloc 29
3.8.2 cudaFree 29
3.8.3 cudaMemcpy 30
3.8.4 cudaMemcpyToSymbol 30
3.8.5 cudaMemset 30
3.8.6 tex1D and tex1Dfetch 30
3.8.7 cudaBindTexture 31
3.9 Compiling and Executing the modified source code 31
Chapter 4 Basic Block Tracing 35
4.1 PTX 35
4.2 Identifying Basic-Blocks 36
4. 3 Back-Annotating CUDA source code 36
4.4 Extracting Basic-Block Sequence of Warp 37
4.5 Example of basic block tracing from PTX 38
Chapter 5 Results 44
5.1 Rodinia Benchmarks 44
5.2 Experimental Results 47
Chapter 6 Summary and Conclusions 57
References 59

List of Tables

Table1: CUDA-specific datatypes. 17

Table 2: GPU Execution times and error percentages of benchmarks................. 48

Table 3: CPU configurations used for testing benchmarks. 49

Table 4: Average CPU slowdown across different configurations. 50

Table 5: Execution times of benchmarks for all configurations of *Intel Core i7-920*. 55

Table 6: Execution times of benchmarks for all configurations of *Intel Core i5-5200U*. 56

List of Figures

Figure 1: Overview of work-flow of simCUDA simulation framework................ 3

Figure 2: Execution sequence of CUDA blocks in different GPUs...................... 7

Figure 3: Organization of threads into blocks and blocks into grids. 8

Figure 4: Example of execution sequence of a warp.. 10

Figure 5: Memory hierarchy in GPUs. 12

Figure 6: Sample CUDA Code. 13

Figure 7: CUDA program compilation process using NVCC. 15

Figure 8: Modeling of CUDA-specific datatype ‘dim3’. 17

Figure 9: Transformation of CUDA implicit variables. 19

Figure 10: Modification of kernel-launch and serialization of thread-blocks. 22

Figure 11: Synchronization primitive modification from CUDA source code. ... 24

Figure 12: Transformation of CUDA shared variable.. 27

Figure 13: Example of CUDA source modification... 34

Figure 14: Example of basic block tracing of a CUDA source code. 43

Figure 15: Slowdown of benchmarks on *Intel Core i7-920* with respect to GPU.50

Figure 16: Slowdown of benchmarks on *Intel Core i5-5200U* with respect to GPU. 51

Figure 17: Scaling of CPU performance on *Intel Core i7-920* CPU. 52

Figure 18: Scaling of CPU performance on *Intel Core i5-5200U* CPU............... 53

Chapter 1

Introduction

Parallel computer hardware has gained much traction in recent years, where every computer manufacturer now seeks to provide a multithreaded hardware that exploits parallelism, and encourages software developers to explicitly use the parallelism as much as possible. The development of these parallel architectures has led to extensive use of parallel programming models [1].

One such class of parallel processing units are called GPUs (Graphics Processing Units [2]). CUDA [3] is a programming model introduced in February, 2007 by NVIDIA to empower programmers to directly utilize a GPU's inherent parallelism. This led to a trend of using conventional GPUs not only for graphics processing applications [4], but also for applications demanding massive parallelism like physics simulations [5], computational finance [6], etc. This class of GPUs started being referred as GPGPUs (General Purpose GPUs).

CUDA has been specifically designed in such a manner that it is similar to general programming languages such as C [7]. It exposes three key abstractions to the programmer in the form of a set of language extensions. These key abstractions include:

1. A hierarchy of thread groups,
2. Shared memories, and
3. Barrier synchronization.

The programming model partitions the problem statement into coarse-grained modules which are solved independently in parallel by blocks of threads. Each module is

further divided into fine-grained sub-modules which are solved using threads within a block that run in parallel and can interact with each other.

CUDA has been vastly popular for GPU architectures. In case of multi-core architectures, a comparable model is currently lacking. The only close competitors to the CUDA programming model in case of multi-core architectures include PThreads [8], MPI [9] and OpenMP [10]. This creates a disadvantage for programmers who have invested their time exploiting the benefits of CUDA in order to write a general purpose application. Thus, an extension to the CUDA programming model supporting multi-core architectures will be highly beneficial to transparently port existing applications and exploit the parallelism that the current generation of multi-core processors have.

Although GPUs by themselves consist of many cores and offer massive parallelism, modern superscalar processors like Intel's CPUs [18] offers four cores with faster single-thread performance compared to a GPU, which makes emulating a GPU's functionality on such CPUs feasible.

1.1 SIMCUDA OVERVIEW

CUDA's use of the specialized GPU features restricts the use of CUDA applications to be executed only in GPUs that support the CUDA constructs. The execution of CUDA constructs is currently not supported by CPUs due to lack of adequate hardware. This makes it challenging to port applications written in CUDA to different CPU architectures.

Our simulation framework is based on C++11 threads, with which we try to mimic the execution of fine-grained threads in a block. We achieve porting of CUDA applications by modeling CUDA constructs as functionally equivalent C++ functions.

This enables us to run a CUDA application on different target platforms. The work-flow overview of our functional CUDA simulator is shown in Figure 1. The simulator takes in a CUDA application as an input. We then use regular expressions to modify the source code and replace it with its equivalent C++ model. We also include additional header files to help with the translation and compilation process. The modified source code is then compiled using a C++ host compiler to generate a host-compatible binary.

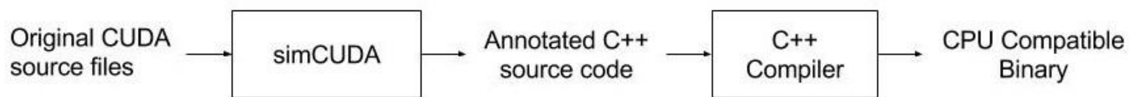


Figure 1: Overview of work-flow of simCUDA simulation framework.

1.2 DESIGN DECISIONS

We chose the GNU Compiler Collection's C++ compiler (G++) as the backend compiler to generate the CPU simulation code. This is because of its large support of the GNU tool-chain and specifically because it is a robust cross compiler which can target most common processor families like ARM , Intel x86, PowerPC, BlackFin, MIPS, SPARC, VAX and Motorola 6800. Thus, G++ is the supported compiler for simCUDA.

This project also required extensive pattern matching and processing of huge amounts of text, which necessitated the need for a scripting language. "Perl" was used as the preferred scripting language for this project because of its wide-spread support, powerful functionality, and ease of use.

1.3 RELATED WORK

In [11], a new detailed microarchitecture performance simulator is designed that runs NVIDIA's parallel thread execution (PTX) virtual instruction set. It accepts the PTX generated from NVIDIA's CUDA compiler (NVCC) as an input and maps them to the current multi-core architecture. Similarly, [12] also uses the PTX thread hierarchy to map to multi-core architectures. This report, instead involves the source-level translation of a CUDA program and does not rely on the PTX ISA. As such, we do not use the NVIDIA CUDA Compiler for the functional model. Nevertheless, NVCC is used in this report to extract the basic-block information from the PTX debug information, which is used to back-annotate the CUDA source code with equivalent tracing support. Source-to-source translation is also used in [13], which uses PThreads to achieve parallelism. They rely on modifying the nature of the kernel function from a per-thread code specification to a per-block code specification, enforcing synchronization with deep fission and replicating the thread-local data, causing an additional overhead. This report rather uses C++11 threads to achieve synchronization between threads, thus eliminating the need to replicate thread-local data. C++11 thread barriers are created and used as synchronization primitives at the thread-level. The work done in this report maintains the kernel-function's nature as a per-thread specification. In doing so, we modify the nature of the kernel call in such a manner that all threads in a block run in parallel, but each block runs serially in a loop. This is because threads in a particular block can communicate with each other and potentially require synchronization, while, on the other hand, threads in different blocks do not communicate and are independent of each other. Thus, it is natural to treat and run each block separately, since it does not involve threads from other blocks. This in turn does not compromise on the accuracy of the process.

1.4 REPORT OUTLINE

The rest of the report is organized as follows: Chapter 2 details the CUDA programming model and analyzes how parallelism is exploited in CUDA. It also details the CUDA architecture and the different stages of compilation that leads to the generation of the binary for the GPU. Chapter 3 provides the implementation details of simCUDA framework. It also outlines how C++11 threads are used to mimic the threads that run in a block in a GPU. Chapter 4 discusses simCUDA's extended functionality of basic block tracing from the intermediate representation of the CUDA source code. Chapter 5 discusses the results of the functional simulator for various benchmarks and how they compare with the results from an actual GPU. Chapter 6 discusses the summary and conclusion drawn from the results of the project, as well as the future work that can be done with the current simulator model.

Chapter 2

CUDA Programming Model

This chapter describes the CUDA programming model, which is based on a *kernel* function that is data-parallel and can be considered as the primary parallel construct [7]. Whenever a kernel is invoked, it creates many CUDA logical threads that are supposed to run in parallel. The threads are further organized in a multi-dimensional format such that a specific number of threads constitute a block. Threads within a block have synchronization primitives, and are free to share data among themselves. But blocks do not have any communication with other blocks and are independent of each other. Thus, no synchronization primitive is needed at the block level. This section gives a high level description of how a CUDA program is executed in a GPU through its massive hardware-level parallelism support.

2.1 KERNELS

Kernels are the basis of parallelism in the CUDA programming model. Simply put, a kernel is a function. It is defined using the *__global__* declaration specifier [7]. When a kernel is invoked or called, it is executed by a specified number of threads, all running in parallel.

2.2 THREADS

The basic unit of execution is a thread. Each thread goes through the kernel function and is free to take its own distinct control path based on the values of its predicates. The thread in a particular block is identified with *threadIdx* which is an

implicitly defined variable and is accessible to all threads. Thus, all threads in a particular block have their own distinct thread IDs.

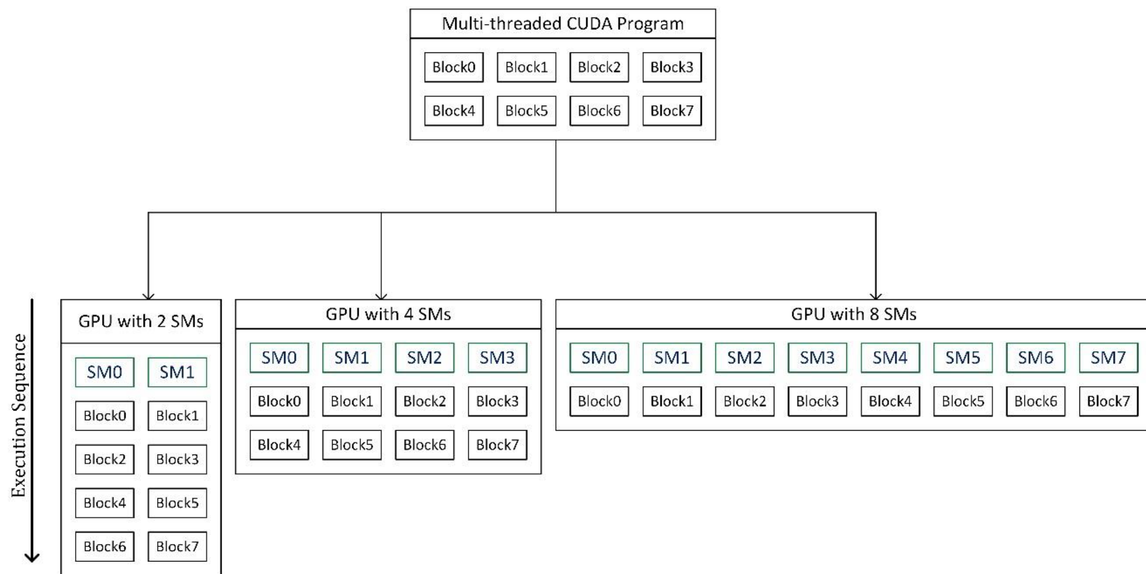


Figure 2: Execution sequence of CUDA blocks in different GPUs.

2.3 BLOCKS

A block is simply a block of threads that share data between them and have synchronization primitives. The implicit variable *threadIdx* is a three-component vector, i.e. it has x, y and z components. Thus, it has the flexibility to form one-dimensional, two-dimensional or three-dimensional thread blocks. Threads in a block reside or execute on the same processor core on a GPU. Thus, the maximum number of threads per block is limited by the memory resources of the core they execute on. The processor core on which the block resides is called a streaming multi-processor (SM). If the number of blocks are more than the number of available SMs, then blocks wait for previous blocks to finish execution on a particular SM before they can execute on the SM. For example, if

a CUDA program is comprised of 8 blocks, then its execution on different GPUs having different number of SMs (2, 4 and 8) is shown in Figure 2 [7]. The number of total threads is given by the number of threads per block multiplied by the total number of thread blocks. Each block is identified with its own implicit variable called *blockIdx* which is built into the CUDA model.

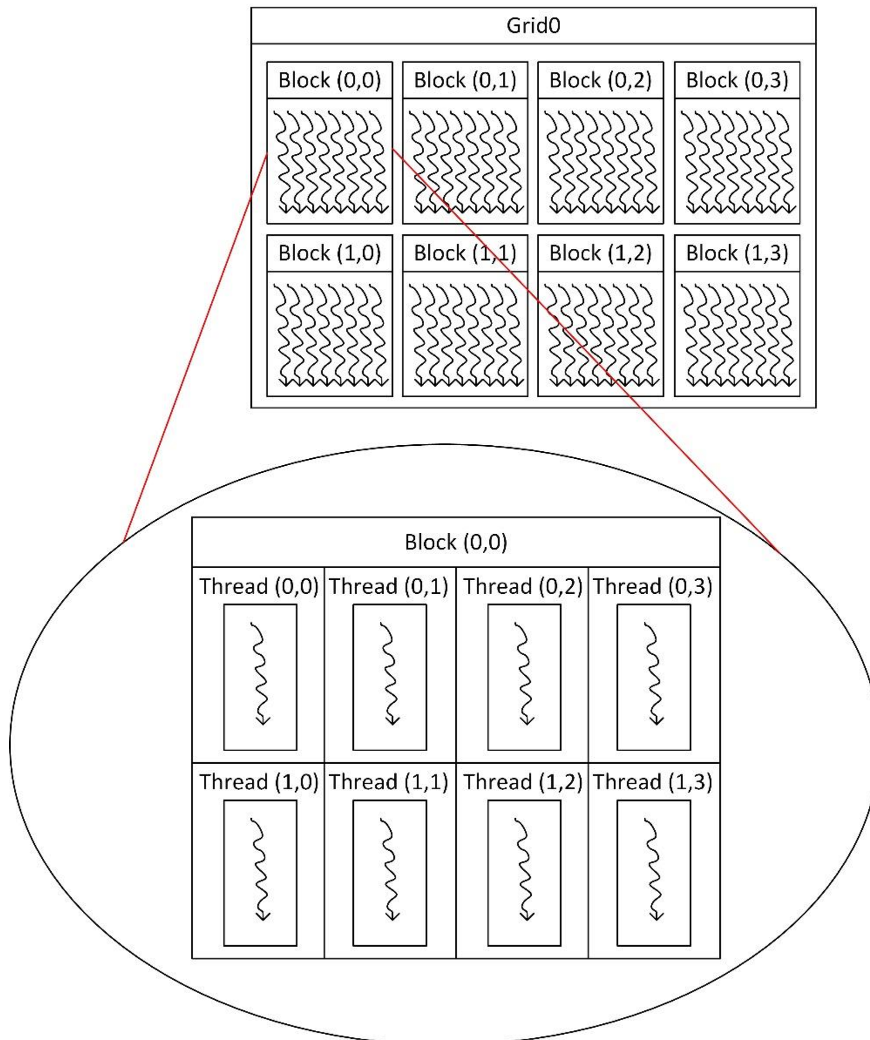


Figure 3: Organization of threads into blocks and blocks into grids.

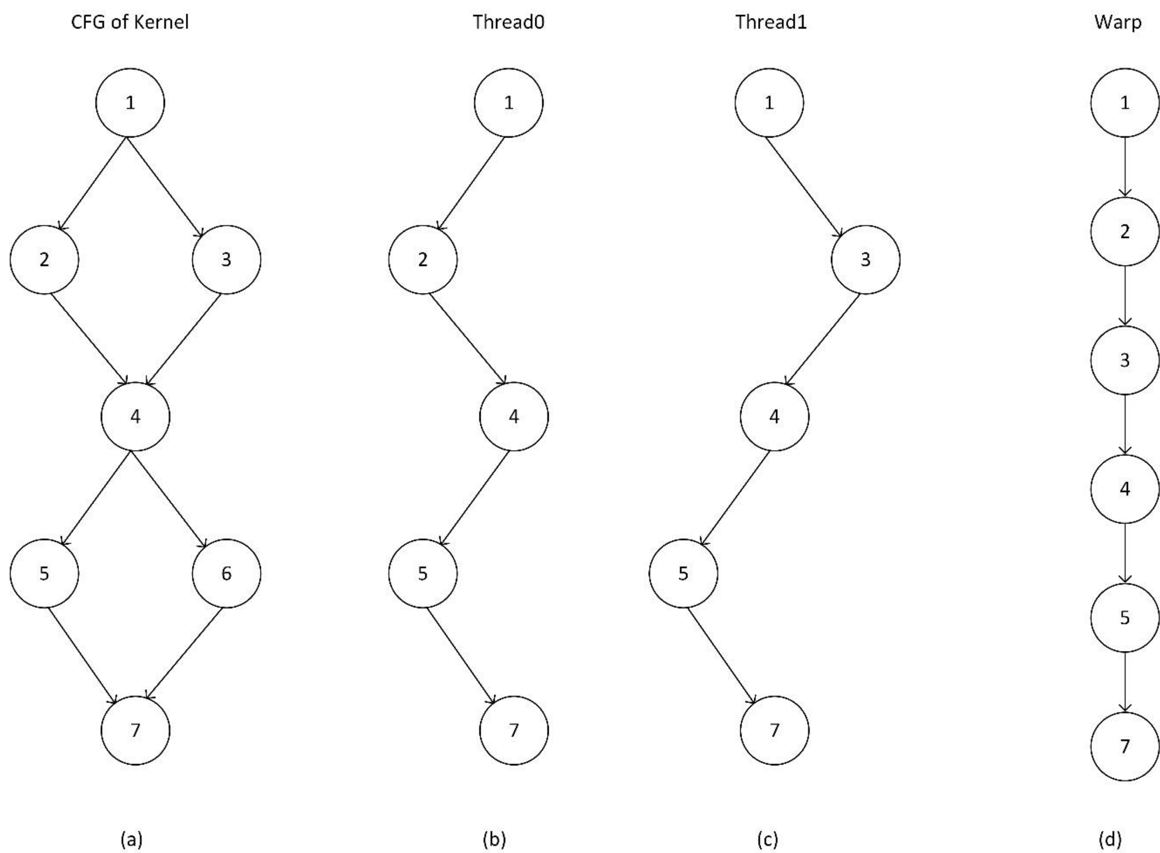
2.4 GRIDS

Blocks are further organized as grids. The grid size is dependent on the number of SMs a GPU has, and is the maximum number of blocks that can run in parallel on the GPU. The number of grids is dependent on the size of the data that is being processed. So, if a GPU has 4 SMs then the grid-size is 4. The organization of a grid into blocks and a block into individual threads is shown in Figure 3. The number of threads in a block or the dimensions of the block is accessible from within the kernel through the built-in variable *blockDim*. Similarly, the dimensions of a grid or the number of blocks per grid is also accessible through the built-in variable *gridDim*.

2.5 WARPS

Each SM splits its own blocks into warps. Threads in the block are actually executed concurrently in groups of maximum 32 threads, which is the size of the warp. Instructions are issued on a per warp basis [16]. Threads in a warp execute in lock-step manner. Thus, the basic block sequence of a warp is useful for calculating the time taken by all the 32 threads inside the warp to execute a particular kernel. An example has been shown in Figure 4, to illustrate this point. Figure 4(a) shows the CFG of a particular kernel which branches at the end of nodes 1 and 4. We assume that only 2 threads are active for a particular warp. The execution sequence of thread-0 is shown in Figure 4(b), while that of thread-1 in warp is shown in Figure 4(c). Since the execution in a warp is done in a lock-step manner, the overall execution sequence is as shown in Figure 4(d). Thus we see that even if thread-1 does not take the branch to node-2, it has to wait for thread-0 to finish executing node-2. Similarly, thread-0 has to wait until thread-1 finishes

executing node-3, and then both of them start executing node-4 in parallel. Thread blocks are thus created in warp-sized units. Also, it is not necessary that at a time all 32 threads of a warp need to be active. It is possible that for a particular kernel any number of threads between 0 and 32 in a warp are active at any given time, while the rest are inactive.



Execution sequence of a warp: (a) CFG of a kernel. (b) Execution sequence of thread-0. (c) Execution sequence of thread-1. (d) Execution sequence of a warp with two active threads (thread-0 and thread-1).

Figure 4: Example of execution sequence of a warp.

2.6 MEMORY

A typical GPU memory hierarchy is shown in Figure 5. The memory hierarchy is divided into 3 types. It consists of:

1. Local memory per thread.
2. Shared Memory per block.
3. Global Memory shared by all blocks.

The local memory per thread is a private local memory that is accessible by each thread. The shared memory is visible to all threads in a block, and all the threads in a block can access as well as change it. The lifetime of the shared memory is the same as the lifetime of the block. The global memory is accessible by all the threads in all the blocks. Apart from these, there are also 2 additional read-only memory types.

1. Constant memory.
2. Texture memory.

As the name suggests, constant memory is used for data that does not change over the course of kernel execution. The constant memory space is cached thus enabling higher bandwidth during read-access. Similar to constant memory, texture memory is also cached on chip, thus providing higher effective bandwidth in some situations. This is done by reducing memory requests to off-chip DRAM. Texture memory is particularly useful for graphics applications where memory access patterns exhibit a great deal of spatial locality. Texture memory is also useful for cases where you update your data rarely, but instead read the data very often. Texture memory has its own data addressing modes, and includes specific data formats.

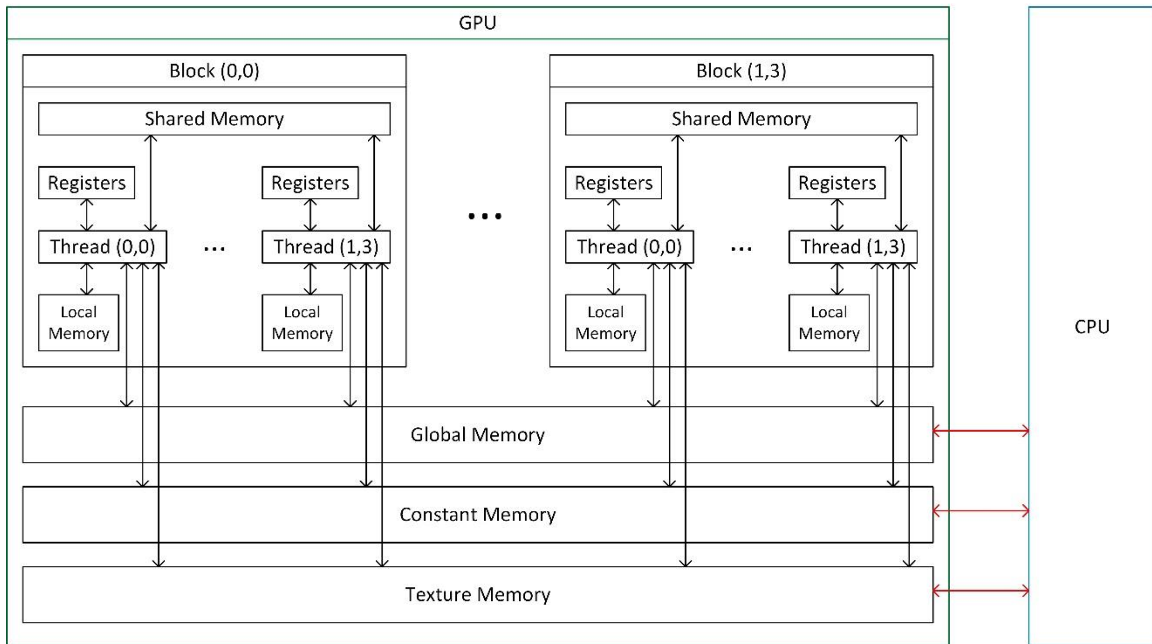


Figure 5: Memory hierarchy in GPUs.

2.7 PROGRAMMING MODEL

As stated before, the kernel is identified by the keyword *__global__* and has access to a number of implicit variables namely, *threadIdx*, *blockIdx*, *blockDim* and *gridDim*. This kernel is called or invoked from a different function simply by stating the function name followed by the syntax <<< … >>>. The variables inside the triple angular brackets represent the number of threads per block, and the total number of blocks. Within a block, each thread has its own local unique ID given by *threadIdx*. The threads also have a global unique ID, given by the formula: ((*blockIdx***blockDim*) + *threadIdx*). Blocks are independent of each other and thus can run in any order, in parallel or in series. The synchronization primitive within a block is the *__syncthreads()* intrinsic

function. It acts as barrier at which all threads in a block must wait. Only when all the threads reach the barrier point are they allowed to proceed further.

A sample CUDA program is shown in Figure 6. The CUDA code in this example reverses an array of 64 integers. The synchronization primitive in the kernel makes sure that the line in the kernel *d[t] = s[tr];* is executed only when all the threads in the block have reached that point. This enables the CUDA program to accurately reverse the array.

```
#include <stdio.h>

__global__ void Reverse(int *d, int n) //Kernel
{
	__shared__ int s[64];
	int t = threadIdx.x;
	int tr = n-t-1;
	s[t] = d[t];
	__syncthreads(); // wait for other threads to reach
	d[t] = s[tr];
}

int main(void)
{
	const int n = 64;
	int a[n], r[n], d[n];

	for (int i = 0; i < n; i++) { //initialise inputs
		a[i] = i;
		r[i] = n-i-1;
		d[i] = 0;
	}

	int *d_d;
	cudaMalloc(&d_d, n * sizeof(int));  // allocate GPU memory
	cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice); //copy data from CPU to GPU

	Reverse<<<1,n>>>(d_d, n); // launch kernel

	cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost); // copy data from GPU to CPU
	for (int i = 0; i < n; i++)
		if (d[i] != r[i])
			printf("Error: d[%d]!=r[%d] (%d, %d)\n", i, i, d[i], r[i]);
	return 0;
 }
```

Figure 6: Sample CUDA Code.

2.8 Compilation and Binary Generation

NVCC is the CUDA compiler which performs the compilation for all CUDA source files [14], [15]. The CUDA compilation process as done by NVCC involves splitting, compilation, preprocessing, and merging steps. CUDA source files are comprised of GPU device functions (extension of C language) on top of a conventional C++ host code. The CUDA compilation flow is responsible for separating the device functions from the host code, and compiling the device functions using the proprietary NVIDIA compilers and assembler. It also compiles the host code using a C++ host compiler, and embeds the compiled GPU functions as fat-binary images into the host object file. CUDA runtime libraries are added during the linking stage to support remote SPMD procedure calling and to provide explicit GPU manipulation, like host GPU data transfers and allocation of GPU memory buffers.

NVCC also accepts compiler options for defining macros, including library paths, and for manipulating the compilation process. The non-CUDA compilation steps are carried out by a C++ host compiler supported by NVCC. The NVCC compiler options are also translated to the appropriate host compiler options internally. The linker at the final stage combines both the CUDA object files from NVCC and the host CPU object files from the host compiler to generate a CPU-GPU executable. The compilation process from CUDA source files to CPU-GPU binaries is shown in Figure 7.

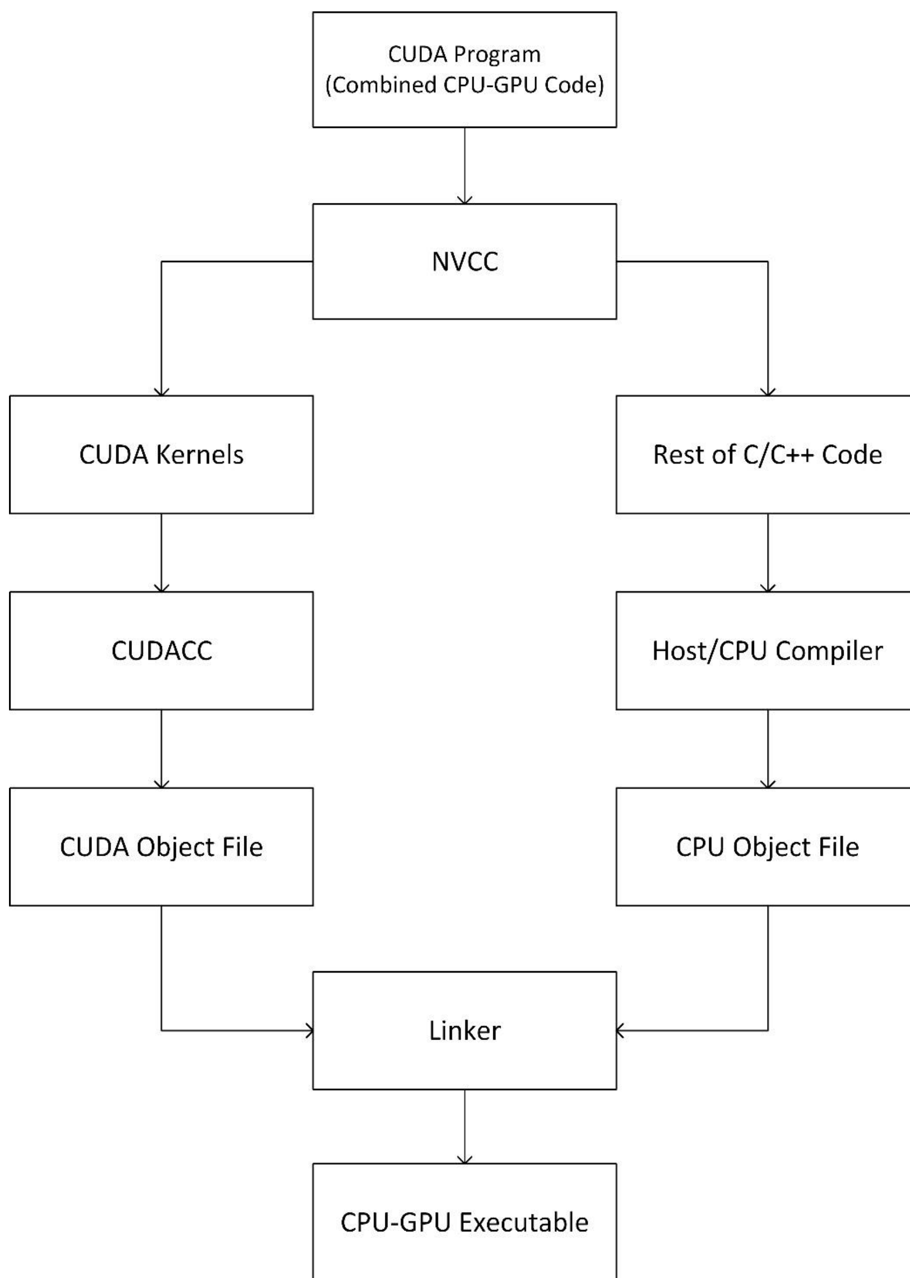


Figure 7: CUDA program compilation process using NVCC.

Chapter 3

Functional Translation of CUDA to C++

The functional simulator uses the source code directly. It modifies the source code in such a manner that it compiles using a host compiler, and thus generates a target binary for a multi-core CPU. Hence, for functional simulation, at no point is the NVIDIA CUDA compiler required. Instead, all that is required is a host compiler that supports C++11 threads (e.g. gcc-4.8), which the simulation process is based on. This chapter describes in detail the translation process, and the specific modifications that a CUDA source code undergoes to compile and run on a host system without the presence of a GPU.

3.1 DEFINING CUDA-SPECIFIC DATATYPES

The CUDA programming model has many datatypes that are not part of the C++ language. These datatypes are an integral part of the CUDA language and are used in a regular fashion. Thus, it is important to define the structures that are part of the CUDA programming model. One such datatype is ‘dim3’. It is a three-component vector type which has three integer components namely x, y and z. We model this datatype using a class as shown in Figure 8 so that the functionality of the datatype remains the same.

Similarly, CUDA has many different datatypes which are shown in Table 1. The number after the datatype represents the number of components of the vector. For example, char2 is a two component vector of type *char* with members x and y. All such datatypes have been modeled in an adequate fashion such that the functionality remains same after the modification process.

```
class dim3
{
public:
        int x, y, z;
        dim3() : x(1), y(1), z(1) {}
        dim3(int a) : x(a), y(1), z(1) {}
        dim3(int a, int b) : x(a), y(b), z(1) {}
        dim3(int a, int b, int c) : x(a), y(b), z(c) {}
};
```

Figure 8: Modeling of CUDA-specific datatype 'dim3'.

Dimensions	**Variable Types**
1	char1, uchar1, short1, ushort1, int1, uint1, long1, ulong1, float1, longlong1, ulonglong1, double1
2	char2, uchar2, short2, ushort2, int2, uint2, long2, ulong2, float2, longlong2, ulonglong2, double2
3	char3, uchar3, short3, ushort3, int3, uint3, long3, ulong3, float3, dim3
4	char4, uchar4, short4, ushort4, int4, uint4, long4, ulong4, float4,

Table1: CUDA-specific datatypes.

3.2 PASSING IMPLICITLY DEFINED VARIABLES TO KERNEL

In CUDA, there are many implicitly defined variables, which serve as identifying features of a thread. Implicitly defined variables generally means that they are built into the language and do not have to be explicitly passed to the kernel. The kernel function can freely access the implicitly defined variables and each thread in each block will have its own copy and value of such variables. Some implicitly defined variables in CUDA include:

1. *threadIdx*
2. *blockIdx*
3. *blockDim*
4. *gridDim*

Unfortunately, C++ does not define any such implicit variables which might help in identifying a particular thread or block. As a result, all these implicitly defined variables have to be explicitly passed, and this is done in the modification process. The modification involves two steps:

Step-1:

This involves changing the signature of the kernel function such that it includes all the implicitly defined variables i.e. *threadIdx*, *blockIdx*, *blockDim* and *gridDim* as function arguments.

Step-2:

This step includes changing the kernel call, wherein apart from the original function arguments we also provide the current thread-ID, block-ID, dimensions of the block, and dimensions of the grid as parameters.

```
__global__ void kernel(int arg)
{
        int thread_id_x = threadIdx.x;
        int block_id_x = blockIdx.x;
        int global_id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
        ...
}
```

(a) CUDA source kernel with implicit variables.

```
void kernel (dim3 blockIdx, dim3 threadIdx, dim3 gridDim, dim3 blockDim, barrier &bar, int arg)
{
        int thread_id_x = threadIdx.x;
        int block_id_x = blockIdx.x;
        int global_id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
        ...
}
```

(b) Modified C++ code with implicit CUDA variables as arguments.

Figure 9: Transformation of CUDA implicit variables.

Figure 9 shows as example of the transformation from CUDA code to C++ code for CUDA implicit variables. Figure 9(a) represents the original CUDA code which has access to all the implicitly defined variables. Figure 9(b) represents the modified C++ code where all the implicitly defined variables are passed as parameters to the kernel function.

3.3 Launching threads

As discussed in Chapter 2, Section 2.7, kernels are launched using triple angular brackets i.e. <<< … >>>. This is specific to CUDA and is not valid for a C++ program. Thus, the modification process is responsible for identifying the kernel launch construct. We do this by using regular expressions in Perl. Once a kernel launch is identified, information about the name of the kernel, the number of threads per block, the number of blocks, and the parameters to the kernel function is extracted from the kernel launch statement. These parameters are then used to launch C++11 threads. The blocks run sequentially in a 'for' loop. The standard process is to define the threads using the syntax of *std::thread* in one iteration of the loop that represents the block. As soon as the thread's associated object is created, the threads start executing, except if there are any scheduling delays by the operating system (OS). We then use *std::thread::join* so that the

main thread waits for all the other threads to finish execution, before it executes the rest of the code.

3.4 SERIALIZATION OF CUDA THREAD BLOCKS

Conceptually, each thread in a GPU is implemented as a single OS thread. However, since the number of threads running in parallel in a GPU is very large, simply allocating an OS thread for each thread in a GPU implies significant overhead. The main challenges include utilization of shared memory only for threads in a block, and the concept of barrier synchronization, which involves synchronization of threads in a block. As, neither the hardware architecture of the host CPU nor the OS can be changed to support the challenges faced, it becomes the responsibility of the translation process to manage the execution of the threads. Since we know that blocks are independent of each other during their execution, and can run in parallel or in series with other blocks, it is natural to run each block serially, and run all the threads in a block in parallel. That way, the compiler only manages the synchronization of all the threads in a particular block at a time. Also, since the blocks are now serialized, all threads communicating with each other during the parallel execution process belong to the same block. This eliminates the risk of communication between threads of different blocks in a natural manner.

Each thread in a block is run as a C++11 thread. The blocks are run in a for-loop, which launches each block one after another serially until all blocks have been executed. Local variables are simply re-used at each loop iteration. Shared variables are also visible across all threads of a block only, and get re-initialized for each new block implemented as a new loop iteration. Also, any synchronization primitive that is being used is

automatically valid for all the threads in a single block, since each iteration only represents a single block.

It is important to note that CUDA allows the number of threads, and blocks to be a three-component vector i.e. both the thread-ID and the block-ID can have x, y and z components. Thus, we modify the CUDA kernel call in a way that only one component of all the threads runs in parallel at a time. As a result, a single kernel call is replaced by a nested loop of overall dimensions of the block which again nests loops over all the components of a thread. This is done mainly to make the job of synchronization between different threads in a block much simpler and easier to track.

An example of modification of the CUDA source code to achieve serialization of blocks, as well as the modifications for a kernel-launch is shown in Figure 10. Figure 10(a) represents the original CUDA source code where a kernel call is made with 'nblocks' number of thread-blocks and 'nthreads' number of threads in each block. Figure 10(b) represents the modified CUDA code as a C++ code. The modified code shows the nested loops for x, y and z components of 'nblocks' which again contain the nested loops for x, y and z components of 'nthreads'.

```
__global__ void kernel (int arg)
{
        ...
}

int main(void)
{
        ...
        dim3 nthreads;
        dim3 nblocks;
        ...
        kernel <<< nblocks, nthreads >>> (arg);
        ...
}
```

(a) CUDA Source Code with a kernel-launch statement.

```
void kernel (dim3 blockIdx, dim3 threadIdx, dim3 gridDim, dim3 blockDim, barrier &bar, int arg)
{
        ...
}

int main(void)
{
                ...
                dim3 threads;
                dim3 blocks;
                ...
/*---------------------------------------- Kernel call for kernel----------------------------------------*/
        dim3 kernel_blocks = blocks;
        std::vector<std::thread> tt_kernel(threads.x);
        barrier bar_kernel(threads.x);

        dim3 kernel_threads = threads;
        for (int block_iz = 0; block_iz < blocks.z; block_iz++) {
        for (int block_iy = 0; block_iy < blocks.y; block_iy++) {
        for (int block_ix = 0; block_ix < blocks.x; block_ix++) {
                dim3 gr_kernel(block_ix,block_iy,block_iz);
                for (int thread_iz = 0; thread_iz < threads.z; thread_iz++) {
                for (int thread_iy = 0; thread_iy < threads.y; thread_iy++) {
                        for (int thread_ix = 0; thread_ix < threads.x; thread_ix++) {
                                dim3 thr_kernel(thread_ix,thread_iy,thread_iz);
                                tt_kernel[thread_ix] = std::thread(kernel,gr_kernel,
                                                                   thr_kernel,kernel_blocks,
                                                                   kernel_threads,std::ref(bar_kernel),arg);
                        }

                        for (int thread_ix = 0; thread_ix < threads.x; thread_ix++)
                                tt_kernel[thread_ix].join();
                }}
        }}}
/*---------------------------------------------- kernel end ----------------------------------------------*/
        ...
}
```

(b) Modified C++ code with serialized blocks and kernel-launch.

Figure 10: Modification of kernel-launch and serialization of thread-blocks.

3.5 SYNCHRONIZATION PRIMITIVE BETWEEN THREADS

C++11 threads [17] provide us with constructs like mutex and condition variables, which are useful tools to create synchronization primitives that work in the host environment. CUDA has an implicit synchronization construct, namely __syncthreads(). This command is a block level synchronization barrier and is used when all threads in a block have to reach a barrier before they are allowed to proceed. A similar barrier is constructed in C++ using condition variables and mutexes, which replicate the functionality of __syncthreads(). Since thread blocks are serialized (as discussed in Section 3.4), we do not have the problem of synchronization barriers affecting threads of other blocks. The member function *std::condition_variable::wait* causes the current

thread to be locked until the condition variable is notified, and is subsequently unblocked when all threads reach the synchronization point and are notified.

```
__global__ void kernel (int arg)
{
        ...
        __syncthreads();
}

int main(void)
{
        dim3 threads, blocks;
        ...
        kernel <<< blocks, threads >>> (arg);
        ...
}
```

(a) CUDA source with synchronization primitive.

```
class barrier
{
public:
        barrier(unsigned int count) : m_threshold(count), m_count(count), m_generation(0)
        {
                if (count == 0)
                        throw std::invalid_argument("count cannot be zero.");
        }

        bool wait()
        {
                //m_mutex.try_lock();
                std::unique_lock<std::mutex> lock(m_mutex);
                unsigned int gen = m_generation;

                if (--m_count == 0)
                {
                        m_generation++;
                        m_count = m_threshold;
                        m_cond.notify_all();
                        return true;
                }

                while (gen == m_generation)
                        m_cond.wait(lock);
                return false;
        }

private:
        std::mutex m_mutex;
        std::condition_variable m_cond;
        unsigned int m_threshold;
        unsigned int m_count;
        unsigned int m_generation;
};
```

(b) Implementation of synchronization barrier in C++

```
 void kernel  (dim3 blockIdx, dim3 threadIdx, dim3 gridDim, dim3 blockDim, barrier &bar, int arg)
{
        ...
        bar.wait();
}

int main(void)
{
        dim3 threads, blocks;
        ...
        barrier bar_kernel(threads.x);
        ...
        tt_kernel[thread_ix] = std::thread(     kernel,gr_kernel,thr_kernel, kernel_blocks,
                                                kernel_threads,std::ref(bar_kernel),arg);
        ...
}
```

(c) Modified C++ code with synchronization barrier.

Figure 11: Synchronization primitive modification from CUDA source code.

In CUDA, the function __syncthreads() is implicitly defined while in the transformed C++ code, the barrier is passed as a parameter. This again involves two steps in the modification process from CUDA to C++ code.

Step 1:

The signature of the kernel function is modified such that it now accepts a barrier object. This barrier object is generally passed by reference so that all the threads in a thread-block have the same synchronization primitive. We call the *wait* function on this barrier object inside the kernel function.

Step 2:

The kernel call is also modified so that a barrier object is first constructed before the kernel call, and then the object is passed by reference to the kernel function using the wrapper *std::ref*, which is basically a value type that behaves like a reference. An example of modification of a synchronization primitive is shown in Figure 11.

3.6 MODIFYING CUDA DEVICE-SPECIFIC AND HOST-SPECIFIC FUNCTIONS

In CUDA, functions can broadly be categorized into three types declared with their own unique keywords as described below:

3.6.1 __global__

The *__global__* keyword is generally used for declaring kernel functions. These are called from the host (CPU) but are executed on the device (GPU). Threads are launched on a kernel function, and thus this type of function mostly marks the entry or starting point of all threads running on the GPU. For the simulator, since there is no actual GPU, it does not make sense to have any distinction as such. Thus, in the modification process, all instances of the *__global__* keyword are removed from the CUDA source files.

3.6.2 __device__

The *__device__* keyword is generally used for declaring device functions. These are called from the device (GPU) and are executed on the device (GPU) as well. Again, for the simulator, since there is no actual device (GPU), the *__device__* keyword has no real significance. Thus, in the modification process this keyword is modeled as an empty string, which does not add any keyword to the existing function. Thus, the function after compilation has the usual signature of a return type followed by the function name and its arguments.

3.6.3 __host__

The *__host__* keyword is used for declaring host functions. In general if a function does not have any declaring keyword, it is assumed to be a host function. These are the functions that are called from the host (CPU) and are executed on the host (CPU) itself. Since C++ has no similar construct and everything is in actuality running on a host machine, this keyword is also modeled as an empty string. Thus, the function after compilation also has the usual signature of a return type followed by the function name and its arguments.

It is worth noting that a function can have both __host__ and __device__ declarations in CUDA. But again, this has no effect on the simulator, since they are both modeled as empty strings.

3.7 MODIFICATION OF MEMORY TYPES

As discussed previously, CUDA has in total 5 different types of memory. Each thread has its own local memory. There is also a shared memory on a per-block basis and a global memory, which is shared across all blocks in all grids. Apart from these, there also exists constant memory and texture memory. Local memory is similar in concept in both GPUs and CPUs i.e. in a CPU running C++11 threads, each thread has its own local memory. Thus, local memory does not need to be specially modeled. In this section all of the memory types, namely shared memory, constant memory, global memory and texture memory have been discussed.

3.7.1 Shared Memory

This is shared between all threads in a block. It is declared using the keyword __*shared*__ with a variable which declares it in a block's shared memory. The declared shared variable is visible to all the threads in a block, and exists for the lifetime of the block. In the simulator, in order to share a variable between all the threads in a block, the variable is modeled as a *static* variable. This makes sure that the variable is initialized only once, and is visible across all parallel threads in a single block. Also, since blocks have been serialized in the modification process, this prevents the sharing of the variable across blocks. An example of the modification of the source code has been shown in Figure 12.

```
__global__ void kernel (int arg)
{
        ...
        __shared__ int var;
        ...
}
```

(a) CUDA source kernel with a shared variable.

```
void kernel  (dim3 blockIdx, dim3 threadIdx, dim3 gridDim, dim3 blockDim, barrier &bar, int arg)
{
        ...
        static int var;
        ...
}
```

(b) Modified kernel with shared variable modeled as static.

Figure 12: Transformation of CUDA shared variable.

3.7.2 Global Memory

As stated previously, global memory is shared between all threads across all blocks. The keyword __*device*__ declares a variable in the global memory of a device (GPU). It is accessible by all threads across all blocks. This is similar to a global variable in a CPU which is visible across all possible threads. Thus, the previous modeling of __*device*__ as an empty string is still valid for the simulator, and needs no further modification.

3.7.3 Constant Memory

Constant memory is used for data that does not change over the course of kernel execution. The keyword __*constant*__ declares a device variable in constant memory. They are accessible across all threads in all blocks within the lifetime of the application. This is similar to global memory in CPU, and thus the modification process involves declaring all the __constant__ variables in a separate header file, and including the header file wherever the variable is used. This way, all threads have access to that variable across all blocks and it is alive for the duration of the application. In order to read from and write to constant memory, CUDA uses constructs like *cudaMemcpyToSymbol* and *cudaMemcpyFromSymbol.* These will be discussed later in this Chapter.

3.7.4 Texture Memory

Texture memory is another type of memory in the CUDA programming model which is particularly useful for graphics applications. It is useful for data that exploits spatial locality, and also for data that is updated rarely but read very often. The various

operations associated with texture memory and their modeling for the simulator will be described in Section 3.8.

3.8 MODIFYING MEMORY OPERATIONS

The CUDA programming model adds its own memory management functions on top of the existing functions used in C for memory operations. These functions are used to transfer data from the host (CPU) to the device (GPU), and also from the device to the host. In case of the simulator, since a device (GPU) is not considered to be in existence, these memory operations have been modeled so that they reflect transfer of data from the host to the host itself. This is not ideal since it wastes some memory space. In the future, the simulator might be optimized further to save memory space and not perform any memory transfer at all. This section describes the CUDA-specific memory operations and how they have been modeled.

3.8.1 cudaMalloc

This function is used to allocate memory of specified bytes in the device and returns a pointer to the newly allocated memory. This is simply modeled as *malloc,* which allocates the same amount of memory in host.

3.8.2 cudaFree

This function is used to free the memory space pointed to by a pointer. Since it behaves in a similar fashion as *free* in C, it has been modeled as such. The memory space in the host previously allocated via *cudaMalloc* is freed by calling the native *free()*.

3.8.3 cudaMemcpy

This function copies a specified number of bytes from a source to a destination. There are four possible directions in which the copy can take place: host-to-host, host-to-device, device-to-host and device-to-device. This too is specified as an argument to the function. For the simulator, since there is no distinction between host and device, the direction of memory transfer becomes irrelevant. Thus, this is modeled as *memcpy* that transfers a fixed size of memory from a source to destination. The fourth argument about the direction of transfer is ignored in this case.

3.8.4 cudaMemcpyToSymbol

This function is generally used to copy data from the host to the device's constant memory. It can also be used to transfer to the device's global memory. Thus, there are only two directions in which the data transfer can take place: host-to-device and device-to-device. For the simulator, this is again modeled using *memcpy* since there is no device, and direction of the transfer is irrelevant.

3.8.5 cudaMemset

This function fills the first specified number of bytes of memory pointed to by a specified pointer with a specified value. Since this is similar to *memset* in C, it has been used to model this function in the simulator.

3.8.6 tex1D and tex1Dfetch

A texture reference defines which part of texture memory is being fetched. It can have an attribute with respect to dimensions. Thus, a texture can be addressed as a one-

dimensional array using one texture co-ordinate, as a two dimensional array with two texture coordinates, and as a three-dimensional array using three texture coordinates. This is represented as *tex1D, tex2D* and *tex3D* respectively. For our purpose, we can simply represent an array as an array itself. So *tex1D* is modeled as a simple array.

tex1Dfetch (texref, i) returns the (i-1)th element from the array texref. This is mostly used to read from the texture memory. Thus, this function is modeled simply as a function returning texref [i] in case of the simulator.

3.8.7 cudaBindTexture

This function is used to bind memory. The *cudaBindTexture* function specifically binds the specified texture reference of specified size to a specific memory area on device pointed to by a specified pointer. For the simulator, since there is no device, there is no texture memory. Thus, this is simply modeled as *memcpy* that transfers the specified size of bytes from the array (texture reference) to the memory location specified by the pointer. If *cudaBindTexture* is used successively on the same pointer, it is supposed to overwrite the original data, which is similar in function to *memcpy* as well.

3.9 COMPILING AND EXECUTING THE MODIFIED SOURCE CODE

The compilation process for CUDA source code has already been discussed in Chapter 2. Once the CUDA source code is translated, we use a normal C++ compiler as the host compiler. For our purpose, we have used g++ version-4.8 since it is compatible with C++11 threads. The compilation process is similar to that of the CUDA compiler with the exception that instead of using NVCC, we use *g++ -std=c++11 –pthread*. The *std* switch is used to specify the standard that the input source files have to follow. The

pthread switch is used to enable the usage of *std::threads,* which are essential for the simulator's multithreading functionality. Apart from this, instead of the CUDA's default include path, we have to specify our own include path that contains the necessary header files for the source code to compile and run.

Once the modified source is compiled, g++ generates an executable capable of running on the host machine. We simply have to run the executable along with suitable arguments (if any), and the CUDA source code should run successfully, and give the same functionality as it would on an actual GPU. An example of the completely modified CUDA source code, its compilation, and console output of an example CUDA source code in shown in Figure 13. Figure 13(a) shows the original CUDA source code. Figure 13(b) shows the modified (converted) C++ code, and Figure 13(c) shows the compilation procedure and the console output of the modified code.

```
/* example.cpp
* host refers to CPU and device refers to GPU `*/

#include <stdio.h>
#include <cuda.h>

// Kernel that executes on the CUDA device
__global__ void squareIt(float *a)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  a[idx] = a[idx] * a[idx];
}

// main routine that executes on the host
int main(void)
{
        float *a_h, *a_d;  // Pointer to host and device array
        a_h = (float *)malloc(16 * sizeof(float));        // Allocate array on host
        cudaMalloc((void **) &a_d, 16 * sizeof(float));   // Allocate array on device

        // Initialize host array of 16 elements
        for (int i=0; i<16; i++) a_h[i] = (float)i;

        //Copy the host array to device
        cudaMemcpy(a_d, a_h, 16 * sizeof(float), cudaMemcpyHostToDevice);

        // Define number of threads and blocks
        int number_of_threads = 4; int number_of_blocks = 4;

        // Call Kernel
        squareIt <<< number_of_blocks, number_of_threads >>> (a_d);

        // Retrieve result from device and store it in host array
        cudaMemcpy(a_h, a_d, sizeof(float)*16, cudaMemcpyDeviceToHost);

        // Print results
        for (int i=0; i<16; i++) printf("%d %f\n", i, a_h[i]);

        // Cleanup memory after use
        free(a_h); cudaFree(a_d);
        return 0;
}
```

(a) Sample CUDA Source code.

```
/* Example.cpp */

#include <cudatypes.h>
#include <stdio.h>

 void square (dim3 blockIdx, dim3 threadIdx, dim3 gridDim, dim3 blockDim, barrier &bar, int* a) //Kernel executing on GPU
{
        int id = (blockIdx.x * blockDim.x) + threadIdx.x;
        a[id] = a[id] * a[id];
}

int main(void)  //main routine executing on host CPU
{
        int *a_host, *a_dev; //Pointer to host and device array
        a_host = (int*) malloc (16 * sizeof(int));      //Allocate array on host
        cudaMalloc(a_dev, 16 * sizeof(int));    //Allocate array on device

        //Initialize host array
        for (int i = 0; i < 16; i++) a_host[i] = (int)i;

        //Copy host array to device
        cudaMemcpy(a_dev, a_host, 16 * sizeof(int), cudaMemcpyHostToDevice);

        //Define number of threads and blocks
        int number_of_threads = 4; int number_of_blocks = 4;

        // Kernel call from host to device
/*------------------------------------------------------------ Kernel call for square-----------------------------------------*/

        dim3 square_number_of_blocks(number_of_blocks);
        std::vector<std::thread> tt_square(number_of_threads);
        barrier bar_square(number_of_threads);

        dim3 square_number_of_threads(number_of_threads);
        for (int block_i = 0; block_i < number_of_blocks; block_i++) {
                dim3 gr_square(block_i);
        for (int thread_i = 0; thread_i < number_of_threads; thread_i++) {
                dim3 thr_square(thread_i);
                tt_square[thread_i] = std::thread(square,gr_square,thr_square,square_number_of_blocks,square_number_of_threads,std::ref(bar_square),a_dev);
                }

        for (int thread_i = 0; thread_i < number_of_threads; thread_i++)
                tt_square[thread_i].join();

        }
/*------------------------------------------------------ square end -----------------------------------------------------------------*/

        //Retrieve result from device and store it back in host array
        cudaMemcpy(a_host, a_dev, 16 * sizeof(int), cudaMemcpyDeviceToHost);

        //Print results
        for (int i = 0; i < 16; i++) printf("%d\t%d\n", i, a_host[i]);

        //Clean-up memory after use
        free(a_host);
        cudaFree(a_dev);
        return 0;
}
```

(b) Modified C++ code of CUDA source code.

```
abhishek@abhishek-ThinkPad-E450:~/PThreads/SimulatorTest/final.bbSeq/simulator_files$ g++ -std=c++11 -pthread -lm -I../../include example.cpp
abhishek@abhishek-ThinkPad-E450:~/PThreads/SimulatorTest/final.bbSeq/simulator_files$ ./a.out
0       0
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100
11      121
12      144
13      169
14      196
15      225
```

(c) Compiling and running the modified C++ code.

Figure 13: Example of CUDA source modification.

Chapter 4

Basic Block Tracing

Basic blocks [19] are straight lines of code that have a single entry and a single exit point with no branches in between. Any program can be divided into a number of basic blocks. In our case, basic block information is important to re-trace how the program would have executed on a real device (GPU). The sequence in which the basic blocks are executed in a program for a given input can be used to estimate and determine the execution time of the program on the GPU without the need for an actual device. simCUDA has an extended functionality for identifying basic blocks. This chapter describes how the basic blocks are identified from PTX Intermediate Representation (IR) of a CUDA source code. Once the basic blocks are identified, they can be used to back-annotate the source code, which then prints out the basic block sequence of all the threads combined when being simulated [20]. Since this represents a large dataset, and all threads in a warp actually run in a lock-step manner, the more important information is how basic-blocks are executed in a warp rather than on a thread-by-thread basis. Since there is no concept of warps in a CPU, this chapter also describes how the basic-block sequence of a warp is computed from the basic block sequence of individual threads.

4.1 PTX

PTX is a virtual GPU instruction set that is by design, independent of the underlying architecture. The PTX achieves interoperability between architectures through a hierarchical model of parallelism, and a set of constructs that ease translation.

PTX itself is very similar to an intermediate-representation of a compiler, which makes the process of identifying useful information and translating them across architectures relatively simple. PTX is used as an IR in our case to identify the basic blocks. Since there are options to get the debug information from the PTX, we use this debug information to back-annotate the source code with the identified basic-blocks.

4.2 IDENTIFYING BASIC-BLOCKS

The IR of the CUDA source code in the form of PTX is used to identify the basic blocks of the kernel. The point of interest is the basic block sequence executed by the kernel, since that is the part of the CUDA code that runs on the device. From the PTX, the basic blocks are identified. Basic block identification uses simple regular expression matching. If a statement is a branch statement or the statement is the start of a function, it marks the entry-point of a basic block. A second branch statement or the end of a function marks the exit-point of a basic block. Also, if a branch target is in between two branches then the basic block is split, where the branch target is considered as the entry point to the new basic block. If the above methods are repeated for the entire PTX code of the CUDA source code, we can easily identify the number of basic blocks and the basic block boundaries. We use Perl to identify the basic blocks from PTX using regular expressions.

4. 3 BACK-ANNOTATING CUDA SOURCE CODE

When the PTX is dumped from a CUDA source code, NVCC has the option of dumping the debug information as well. This debug information in the PTX code provides us with the information about the location of the files that are part of the PTX

code, as well as the line-number of the line which PTX code segment represents. Again, Perl is used to extract the file-location and the line-number of the PTX code, and whenever a basic block boundary is reached we annotate the corresponding line-number in the corresponding file with the basic block number. Each kernel has its own basic block numbers and boundaries. The back-annotation generally includes writing the thread-ID, along with the basic block number(s) to a file. This way, the basic block sequence of all the threads that were active at some point of time get dumped to a file, from where they can be processed further.

4.4 Extracting Basic-Block Sequence of Warp

Chapter 2 describes how each thread inside a warp executes in a lock-step fashion. This makes finding the basic block sequence of execution on a per-warp basis much more important and useful than on a per-thread basis. Unfortunately, the host does not have any concept of warps as such, but we do have data for the basic block sequence of execution of all threads. Generally, 32 threads constitute a warp, and warps still exist in a block. Thus, all the threads in a block can be grouped, where each group of 32 constitutes a warp. This is the basis for modeling a warp in simCUDA. In order to make a group of 32 threads model a lock-step execution, the idea is to take the union of all the 32 threads in their basic block execution sequence based on the basic block number. We use Perl to recursively take the union of 32 threads, which gives the lock-step sequence in which the basic blocks of all the threads in a single warp execute.

4.5 EXAMPLE OF BASIC BLOCK TRACING FROM PTX

Figure 14 walks through the complete basic block tracing procedure using simCUDA. Figure 14(a) shows an example of a CUDA source code with branching statements. It computes the number of odd and even elements in an array, as well as the number of elements greater than *limit* and less than or equal to *limit* in an array.

NVCC is then used to generate the PTX for the CUDA source code. The corresponding PTX for the CUDA source code is shown in Figure 14(b). The basic block information is identified as discussed in Section 4.2. The debug information from the PTX includes specific files which have to be annotated, given by the file number, and the exact line number which corresponds to a part of the PTX code. The *.loc* keyword in the PTX denotes the file number and the line number of the source file to which the PTX segment belongs. From the PTX, seven basic blocks are identified in this example, as highlighted in yellow in the figure. Their corresponding line numbers are highlighted in green. The summary of the basic block boundaries is shown in Figure 14(c). This is the output of simCUDA extension that shows the basic block numbers, their corresponding line numbers and the files that are annotated.

This information is used to find a suitable place for annotating the source code with the basic block information. Figure 14(d) shows the back-annotated CUDA source code with the basic block information. As can be seen from the figure, the basic blocks are annotated at the nearest valid location of the line-number extracted from the PTX.

Once we get the back-annotated CUDA source code, we use simCUDA's functional framework to generate the corresponding C++ code. The modified C++ code is then compiled using a host compiler (g++-4.8). This generates a file that contains the basic block execution sequence for each thread. This file containing the execution sequence of each thread is post-processed as discussed in Section 4.4 to generate the

basic block execution sequence of the warp. The basic block execution sequence for each of the threads derived from the back-annotated CUDA source code, and the corresponding execution sequence of a warp computed by taking the union of all the threads is shown in Figure 14(e).

```
1 #include <iostream>
2 __global__ void categorize( int* a, int* p )
3 {
4         int tid = threadIdx.x;
5         if ( a[tid] % 2 == 0 )
6         {
7                 p[0]++;
8         }
9         else
10        {
11                p[1]++;
12        }
13        int limit = 5;
14        if ( a[tid] > limit )
15        {
16                p[2]++;
17        }
18        else
19        {
20                p[3]++;
21        }
22 }
23
24 int main(void) {
25        int h_a[10]; int nthreads = 10; int nblocks = 1; int* d_a; int *d_m;
26        int param[4] = {0,0,0,0}; // categories to be updated
27        for (int i = 0; i < 10; i++)  h_a[i] = i;
28        cudaMalloc ((void**)&d_a, sizeof(int)*10);
29        cudaMalloc ((void**)&d_m, sizeof(int)*10);
30        cudaMemcpy( d_a, h_a, 10*sizeof(int), cudaMemcpyHostToDevice );
31        cudaMemcpy( d_m, param, 4*sizeof(int), cudaMemcpyHostToDevice );
32        categorize <<< nblocks, nthreads >>> ( d_a, d_m );
33        cudaMemcpy( param, d_m, 4*sizeof(int), cudaMemcpyDeviceToHost );
34        for (int i = 0; i < 4; i++) std::cout << param[i];
35        return 0;
36 }
37
```

(a) Sample CUDA source code.

```
        .file   1       "<command-line>"
        .file   2       "/tmp/tmpxft_00000faa_00000000-8_example4.cudafe2.gpu"
        .file   3       "/usr/lib/gcc/i686-linux-gnu/4.4.7/include/stddef.h"
        .file   4       "/usr/local/cuda/bin/../include/crt/device_runtime.h"
        .file   5       "/usr/local/cuda/bin/../include/host_defines.h"
        .file   6       "/usr/local/cuda/bin/../include/builtin_types.h"
        .file   7       "/usr/local/cuda/bin/../include/device_types.h"
        .file   8       "/usr/local/cuda/bin/../include/driver_types.h"
        .file   9       "/usr/local/cuda/bin/../include/surface_types.h"
        .file   10      "/usr/local/cuda/bin/../include/texture_types.h"
        .file   11      "/usr/local/cuda/bin/../include/vector_types.h"
        .file   12      "/usr/local/cuda/bin/../include/device_launch_parameters.h"
        .file   13      "/usr/local/cuda/bin/../include/crt/storage_class.h"
        .file   14      "/usr/include/i386-linux-gnu/bits/types.h"
        .file   15      "/usr/include/time.h"
        .file   16      "example4.cu"
        .file   17      "/usr/local/cuda/bin/../include/common_functions.h"
        .file   18      "/usr/local/cuda/bin/../include/math_functions.h"
        .file   19      "/usr/local/cuda/bin/../include/math_constants.h"
        .file   20      "/usr/local/cuda/bin/../include/device_functions.h"
        .file   21      "/usr/local/cuda/bin/../include/sm_11_atomic_functions.h"
        .file   22      "/usr/local/cuda/bin/../include/sm_12_atomic_functions.h"
        .file   23      "/usr/local/cuda/bin/../include/sm_13_double_functions.h"
        .file   24      "/usr/local/cuda/bin/../include/sm_20_atomic_functions.h"
        .file   25      "/usr/local/cuda/bin/../include/sm_20_intrinsics.h"
        .file   26      "/usr/local/cuda/bin/../include/surface_functions.h"
        .file   27      "/usr/local/cuda/bin/../include/texture_fetch_functions.h"
        .file   28      "/usr/local/cuda/bin/../include/math_functions_dbl_ptx1.h"


        .entry _Z10categorizePiS_ (
                .param .u32 __cudaparm__Z10categorizePiS__a,
                .param .u32 __cudaparm__Z10categorizePiS__p)
        {
        .reg .u16 %rh<3>;
        .reg .u32 %r<20>;
        .reg .pred %p<4>;
        .loc    16      2       0
$LDWbegin__Z10categorizePiS_:
        cvt.s32.u16     %r1, %tid.x;
        cvt.u16.u32     %rh1, %r1;
        mul.wide.u16    %r2, %rh1, 4;
        ld.param.u32    %r3, [__cudaparm__Z10categorizePiS__a];
        add.u32         %r4, %r3, %r2;
        ld.param.u32    %r5, [__cudaparm__Z10categorizePiS__p];
        ld.global.s32   %r6, [%r4+0];
        and.b32         %r7, %r6, 1;
        mov.u32         %r8, 0;
        setp.ne.u32     %p1, %r7, %r8;
        @%p1 bra        $Lt_0_2050;
        ld.param.u32    %r5, [__cudaparm__Z10categorizePiS__p];
        .loc    16      7       0
        ld.global.s32   %r9, [%r5+0];
        add.s32         %r10, %r9, 1;
        st.global.s32   [%r5+0], %r10;
        bra.uni         $Lt_0_1794;
```

(b) Corresponding PTX IR for CUDA source code.

```
$Lt_0_2050:
        ld.param.u32    %r5, [__cudaparm__Z10categorizePiS__p];
        .loc    16      11      0
        ld.global.s32   %r11, [%r5+4];
        add.s32         %r12, %r11, 1;
        st.global.s32   [%r5+4], %r12;
$Lt_0_1794:
        .loc    16      13      0
        ld.global.s32   %r13, [%r4+0];
        mov.u32         %r14, 5;
        setp.le.s32     %p2, %r13, %r14;
        @%p2 bra        $Lt_0_2562;
        ld.param.u32    %r5, [__cudaparm__Z10categorizePiS__p];
        .loc    16      16      0
        ld.global.s32   %r15, [%r5+8];
        add.s32         %r16, %r15, 1;
        st.global.s32   [%r5+8], %r16;
        bra.uni         $Lt_0_2306;
$Lt_0_2562:
        ld.param.u32    %r5, [__cudaparm__Z10categorizePiS__p];
        .loc    16      20      0
        ld.global.s32   %r17, [%r5+12];
        add.s32         %r18, %r17, 1;
        st.global.s32   [%r5+12], %r18;
$Lt_0_2306:
        .loc    16      22      0
        exit;
$LDWend__Z10categorizePiS_:
        } // _Z10categorizePiS_
```

(b) Corresponding PTX code for CUDA source code (contd.).

```
Basic Block Boundaries:
bb_0_0 (File: /home/abhishek/PThreads/SimulatorTest/final.bbSeq/exampleTest/example4.cu Line: 2)
bb_0_1 (File: /home/abhishek/PThreads/SimulatorTest/final.bbSeq/exampleTest/example4.cu Line: 7)
bb_0_2 (File: /home/abhishek/PThreads/SimulatorTest/final.bbSeq/exampleTest/example4.cu Line: 11)
bb_0_3 (File: /home/abhishek/PThreads/SimulatorTest/final.bbSeq/exampleTest/example4.cu Line: 13)
bb_0_4 (File: /home/abhishek/PThreads/SimulatorTest/final.bbSeq/exampleTest/example4.cu Line: 16)
bb_0_5 (File: /home/abhishek/PThreads/SimulatorTest/final.bbSeq/exampleTest/example4.cu Line: 20)
bb_0_6 (File: /home/abhishek/PThreads/SimulatorTest/final.bbSeq/exampleTest/example4.cu Line: 22)
```

(c) Basic block locations in their corresponding CUDA source file.

```
 1 #include <iostream>
 2 __global__ void categorize( int* a, int* p )
 3 { dump(threadIdx.x + (blockIdx.x * blockDim.x),"bb_0_0");
 4         int tid = threadIdx.x;
 5         if ( a[tid] % 2 == 0 )
 6         {
 7                 p[0]++; dump(threadIdx.x + (blockIdx.x * blockDim.x),"bb_0_1");
 8         }
 9         else
10         {
11                 p[1]++; dump(threadIdx.x + (blockIdx.x * blockDim.x),"bb_0_2");
12         }
13         int limit = 5; dump(threadIdx.x + (blockIdx.x * blockDim.x),"bb_0_3");
14         if ( a[tid] > limit )
15         {
16                 p[2]++; dump(threadIdx.x + (blockIdx.x * blockDim.x),"bb_0_4");
17         }
18         else
19         {
20                 p[3]++; dump(threadIdx.x + (blockIdx.x * blockDim.x),"bb_0_5");
21         }
22 dump(threadIdx.x + (blockIdx.x * blockDim.x),"bb_0_6"); }
23
24 int main(void) {
25         int h_a[10]; int nthreads = 10; int nblocks = 1; int* d_a; int *d_m;
26         int param[4] = {0,0,0,0}; // categories to be updated
27         for (int i = 0; i < 10; i++)  h_a[i] = i;
28         cudaMalloc ((void**)&d_a, sizeof(int)*10);
29         cudaMalloc ((void**)&d_m, sizeof(int)*10);
30         cudaMemcpy( d_a, h_a, 10*sizeof(int), cudaMemcpyHostToDevice );
31         cudaMemcpy( d_m, param, 4*sizeof(int), cudaMemcpyHostToDevice );
32         categorize <<< nblocks, nthreads >>> ( d_a, d_m );
33         cudaMemcpy( param, d_m, 4*sizeof(int), cudaMemcpyDeviceToHost );
34         for (int i = 0; i < 4; i++) std::cout << param[i];
35         return 0;
36 }
37
```

(d) Back-Annotated CUDA source code with basic block information.

```
Thread0:   Thread3:   Thread6:   Thread9:   Warp0:
bb_0_0     bb_0_0     bb_0_0     bb_0_0     bb_0_0
bb_0_1     bb_0_2     bb_0_1     bb_0_2     bb_0_1
bb_0_3     bb_0_3     bb_0_3     bb_0_3     bb_0_2
bb_0_5     bb_0_5     bb_0_4     bb_0_4     bb_0_3
bb_0_6     bb_0_6     bb_0_6     bb_0_6     bb_0_4
                                            bb_0_5
Thread1:   Thread4:   Thread7:              bb_0_6
bb_0_0     bb_0_0     bb_0_0
bb_0_2     bb_0_1     bb_0_2
bb_0_3     bb_0_3     bb_0_3
bb_0_5     bb_0_5     bb_0_4
bb_0_6     bb_0_6     bb_0_6

Thread2:   Thread5:   Thread8:
bb_0_0     bb_0_0     bb_0_0
bb_0_1     bb_0_2     bb_0_1
bb_0_3     bb_0_3     bb_0_3
bb_0_5     bb_0_5     bb_0_4
bb_0_6     bb_0_6     bb_0_6
```

(e) Basic block execution sequence of all threads and the corresponding warp.

Figure 14: Example of basic block tracing of a CUDA source code.

As seen from Figure 14(e), the warp sequence derived by taking the union of all the threads in the warp, represents a lock-step execution. This is consistent with the real-world behavior of a warp executing on a GPU, where all the threads in a warp execute in a lock-step fashion as discussed in Section 2.5.

Chapter 5

Results

We tested our simCUDA simulation framework on a state-of-the-art CUDA benchmark suite. This section first describes the benchmark suite and its general characteristic. The description is followed by a performance evaluation of the simulator in terms of functionality and execution time.

5.1 Rodinia Benchmarks

Rodinia [25] is a benchmark suite for heterogeneous computing, which includes applications and kernels that target multi-core CPU and GPU platforms. For our purposes, we used the applications that were specifically targeted towards an NVIDIA GPU. The applications, after translation and modification through simCUDA, were simulated in a multi-core CPU. The Rodinia benchmark suite has been used since it covers a wide range of data sharing and parallelism characteristics, and also spans across different types of behavior. The diverse applications make it an ideal test-suite since it sheds light on the capabilities of the simulator with respect to the types of applications, amount of data and parallelism that it can support. The benchmarks used to test the simulator are described below:

1. **Back Propagation**: A machine learning algorithm based on a layered neural network. It computes the weights of connecting nodes on the neural network [27].
2. **Breadth-First Search**: This algorithm traverses all the nodes in a graph one level at a time [28].

3. **B+ Tree**: An application with numerous internal commands for maintaining databases and processing queries, of which J and K commands were ported to CUDA and other parallel languages. The current implementation uses the same algorithms for both commands to expose fine-grained parallelism [29].
4. **CFD Solver**: An unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow [30].
5. **Discrete Wavelet Transform**: A library implementing forward and reverse DWT 9/7 and 5/3 transforms [31].
6. **Gaussian Elimination**: For each iteration, values are computed in parallel and then the algorithm synchronizes between each iteration solving the variables of a linear system in a row-by-row fashion.
7. **Heart Wall**: This application tracks the movement of a mouse's heart over a sequence of 104 609x590 ultrasound images recording responses to the stimulus [32]. The benchmark involves three stages – an initial stage involving image processing to detect initial, partial shapes of inner and outer heart walls. Full shapes of heart walls are reconstructed by the program by generating ellipses that are superimposed over the image and sampled to mark points on the heart walls. In its final stage, movement of surfaces is tracked by detecting the movement of image areas under sample points.
8. **HotSpot**: A tool to estimate processor temperature based on an architectural floorplan and simulated power measurements. The benchmark involves re-implementation of a transient thermal differential equation solver [33].
9. **Hybrid Sort**: A sorting procedure involving a parallel bucket-sort that splits the list into enough sub-lists to be sorted in parallel using merge-sort [26].

10. **K-Means**: A clustering algorithm that identifies related points by associating each data point with its nearest cluster. It then computes new cluster centroids, and keeps on iterating until it converges [21].
11. **LavaMD**: An implementation to compute particle potential and relocation due to mutual forces between particles within a large three-dimensional space.
12. **Leukocyte**: An algorithm for tracking moving white blood cells in video microscopy of blood vessels across many frames [22]. The maximal Gradient Inverse Coefficient of Variation (GICOV) of each pixel is computed across a range of ellipses and a Motion Gradient Vector Flow (MGVF) matrix is calculated for the area surrounding each cell.
13. **LUD**: An algorithm to calculate the solutions of a set of linear equations, wherein a matrix is decomposed as the product of a lower triangular matrix and an upper triangular matrix.
14. **Myocyte**: Simulates the behavior of cardiac myocyte according to the work in [32].
15. **Nearest Neighbors**: An algorithm that computes the k-nearest neighbors in an unstructured data-set, based on the Euclidian distance from a given target latitude and longitude.
16. **Needleman-Wunsch**: A non-linear global optimization method for DNA sequence alignment [23].
17. **Particle Filter**: An algorithm that statistically estimates a target object's location from a noisy measurement setup and an idea of the object's path in a Bayesian framework [24].
18. **Shortest Path (Pathfinder)**: Finds a path from the bottom row to the top row of a two-dimensional grid with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally ahead, using dynamic programming.

19. **Speckle Reducing Anisotropic Diffusion**: Based on partial differential equations, it is used to remove locally correlated noise, known as speckles, without destroying important image features [32].
20. **Stream Cluster**: An algorithm for solving the online clustering problem which finds a pre-determined number of medians so that each input point is assigned to its nearest center [34].

5.2 Experimental Results

The benchmarks discussed in the previous section were run on the simulator to compute the error percentages with respect to outputs from the GPU. Table 2 lists the error percentages of the benchmarks running on an *Intel Core i5-5200* CPU compared to the results from a GPU, and the execution times of all the benchmarks on a GPU. For some of the benchmarks, it was not possible to quantify an error percentage since the output produced was a pictorial one, as is the case for benchmarks dwt2d and srad_v1. The error percentages are computed with respect to the same benchmark running on a system with a native GPU. The most plausible explanation of the difference in outputs of the simulator and the system with a native GPU is non-determinism in the order in which threads are scheduled on the CPU. Since the number of threads running in parallel in a CPU is lower and the thread-scheduling is handled by the OS itself, the outputs may have a minor deviation as indicated by the error percentage.

Benchmarks	GPU Execution Time (seconds)	Error Percentage (%)
B+ Tree	0.106	0
Back propagation	0.093	0
BFS	0.173	0
CFD Solver	0.841	0
DWT2D	0.192	(Pictorial Output)
Gaussian	0.125	0
Heart Wall	0.697	0
HotSpot	0.389	0.009317
Hybrid Sort	0.109	0
K-means	1.478	0
LavaMD	0.175	2.714286
Leukocyte	0.139	0.000491
LUD	0.096	0
Myocyte	0.340	0.333077
Nearest-Neighbors	0.103	0
Needleman-Wunsch	0.137	0
Particle filter	0.224	1.576896
Pathfinder	0.993	0
SRAD (v1)	0.108	(Pictorial Output)
SRAD (v2)	0.101	3.463221
Stream Cluster	0.282	0

Table 2: GPU Execution times and error percentages of benchmarks

The benchmarks were executed on an *NVIDIA GeForce GTX TITAN Black* GPU and their execution times were noted. For the execution time of the benchmarks in multi-core CPUs, an *Intel Core i7-920* CPU, an x64-based quad-core processor with hyper-threading functionality, and an *Intel Core i5-5200* CPU, an x64-based dual-core processor with hyper-threading functionality were used. For the tests, 5 different configurations were used across both CPUs as listed in Table 3.

Test Name	CPU Configuration
c4_ht_on	All 4 CPU cores active with Hyper-Threading enabled.
c4_ht_off	All 4 CPU cores active with Hyper-Threading disabled.
c2_ht_on	2 CPU cores active with Hyper-Threading enabled.
c2_ht_off	2 CPU cores active with Hyper-Threading disabled.
c1	Only one CPU core active with no Hyper-Threading

Table 3: CPU configurations used for testing benchmarks.

The execution times of the benchmarks running on an *Intel Core i7-920* CPU normalized against execution times on the GPU for different configurations are shown in Figure 15. Similarly, the comparison of execution times of the benchmarks running on an *Intel Core i5-5200* CPU normalized against the execution times on the GPU for different configurations are shown in Figure 16.

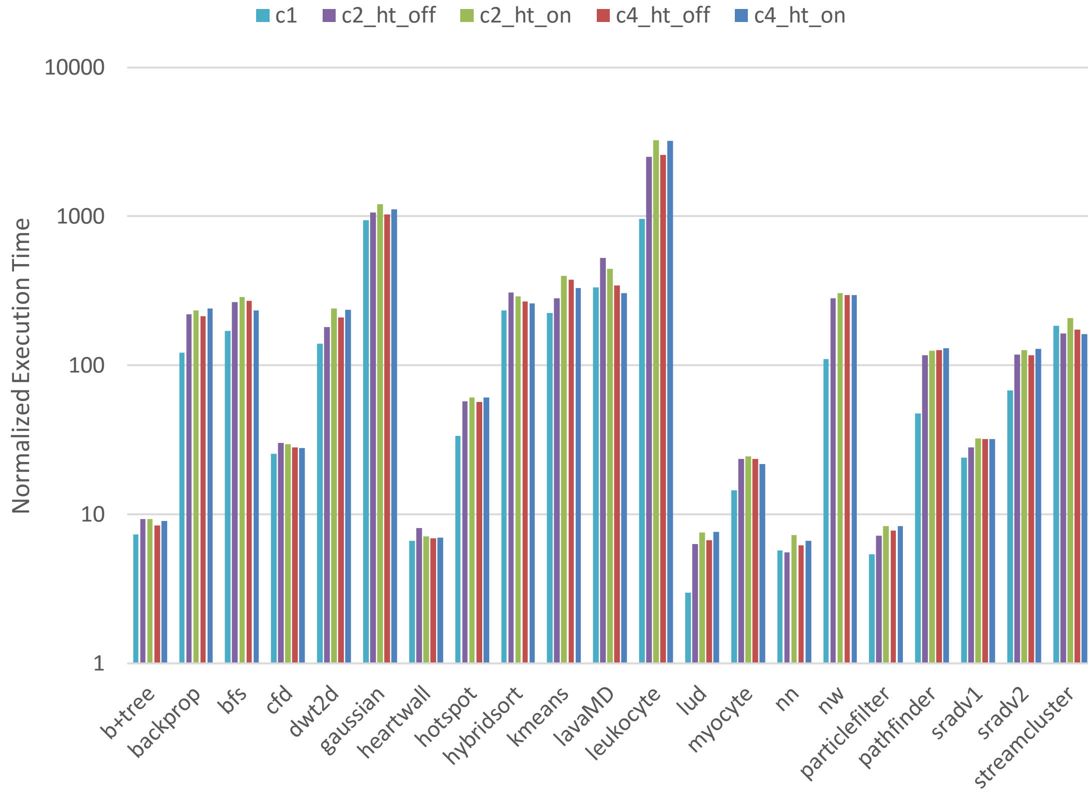


Figure 15: Slowdown of benchmarks on *Intel Core i7-920* with respect to GPU.

CPU Configuration	**Average CPU Slowdown w.r.t GPU for all benchmarks**	
	Intel Core i7-920	*Intel Core i5-5200U*
c4_ht_on	324	Not Applicable
c4_ht_off	295	Not Applicable
c2_ht_on	347	193
c2_ht_off	296	195
c1	175	156

Table 4: Average CPU slowdown across different configurations.

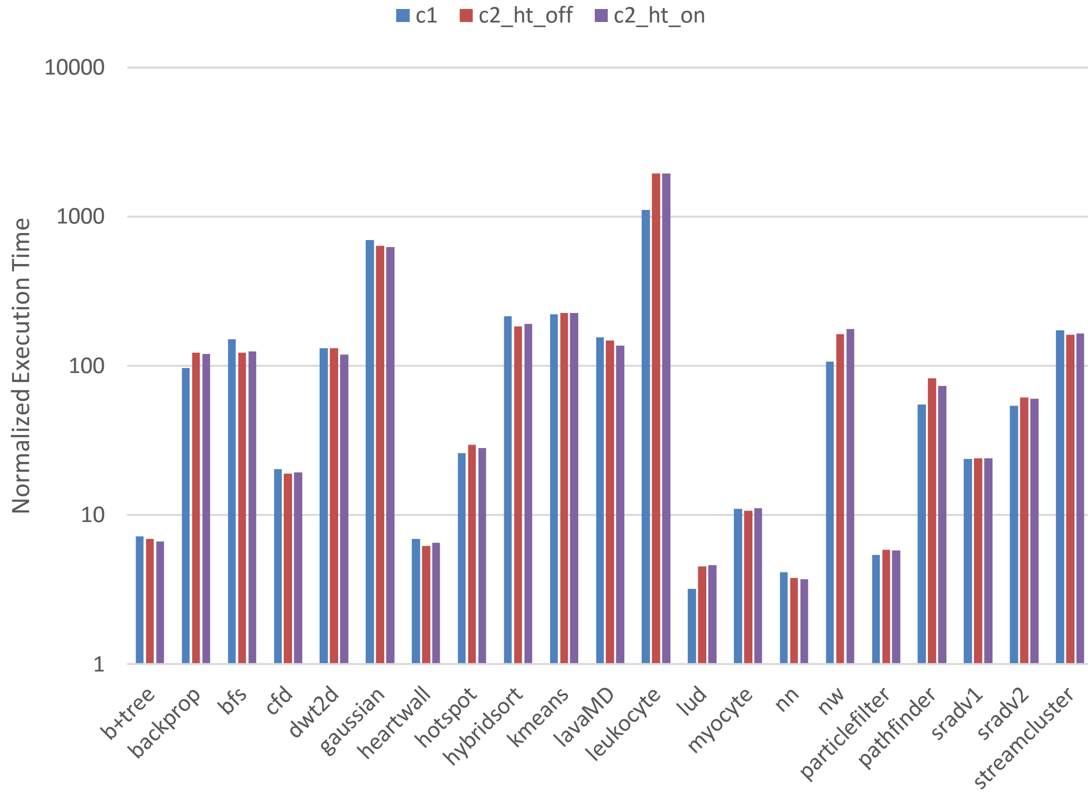


Figure 16: Slowdown of benchmarks on *Intel Core i5-5200U* with respect to GPU.

In general, the average ratio of execution time of a benchmark on an *Intel Core i7-920* CPU to the execution time of the same benchmark on a GPU is 287 for all the test configurations of the CPU. Across all configurations of the *Intel Core i7-920* CPU, the minimum observed slowdown is 3x**,** for the benchmark LUD, when only one core is active and Hyper-Threading is disabled. The maximum observed slowdown is 3230x for the Leukocyte benchmark when 2 cores are active and Hyper-Threading is enabled. The average ratio of execution time of a benchmark on an *Intel Core i5-5200U* CPU to the execution time of the same benchmark on a GPU is 182 for all the test configurations of the CPU. Across all configurations of the *Intel Core i5-5200U* CPU, the minimum observed slowdown is 3.2x**,** again for the benchmark LUD, when only one core is active

and Hyper-Threading is disabled. Similarly, the maximum observed slowdown is 1940x, for the Leukocyte benchmark when 2 cores are active and Hyper-Threading is disabled. The average slowdown for different CPU configurations on both the *Intel Core i7-920* CPU, and the *Intel Core i5-5200U* CPU are shown in Table 4.

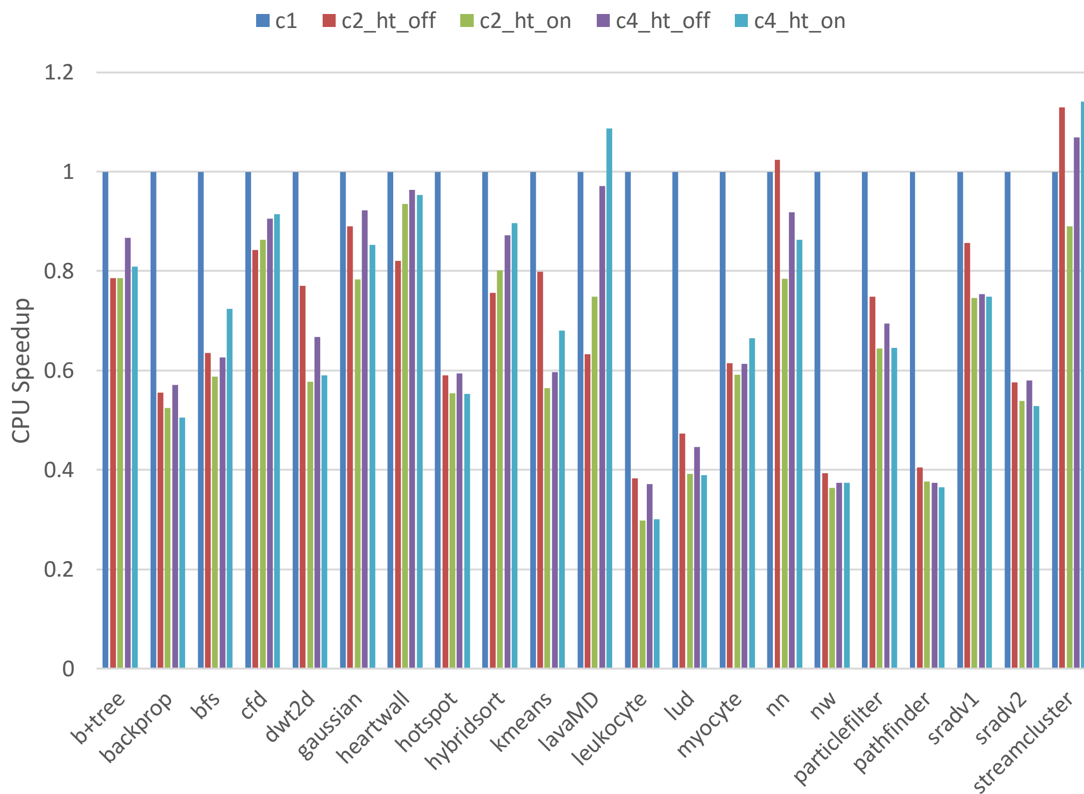


Figure 17: Scaling of CPU performance on *Intel Core i7-920* CPU.

The CPU performance for different configurations were also measured against the same CPU's single active core configuration. The scaling of CPU performance on the *Intel Core i7-920* is shown in Figure 17, while that of *Intel Core i5-5200U* is shown in Figure 18.

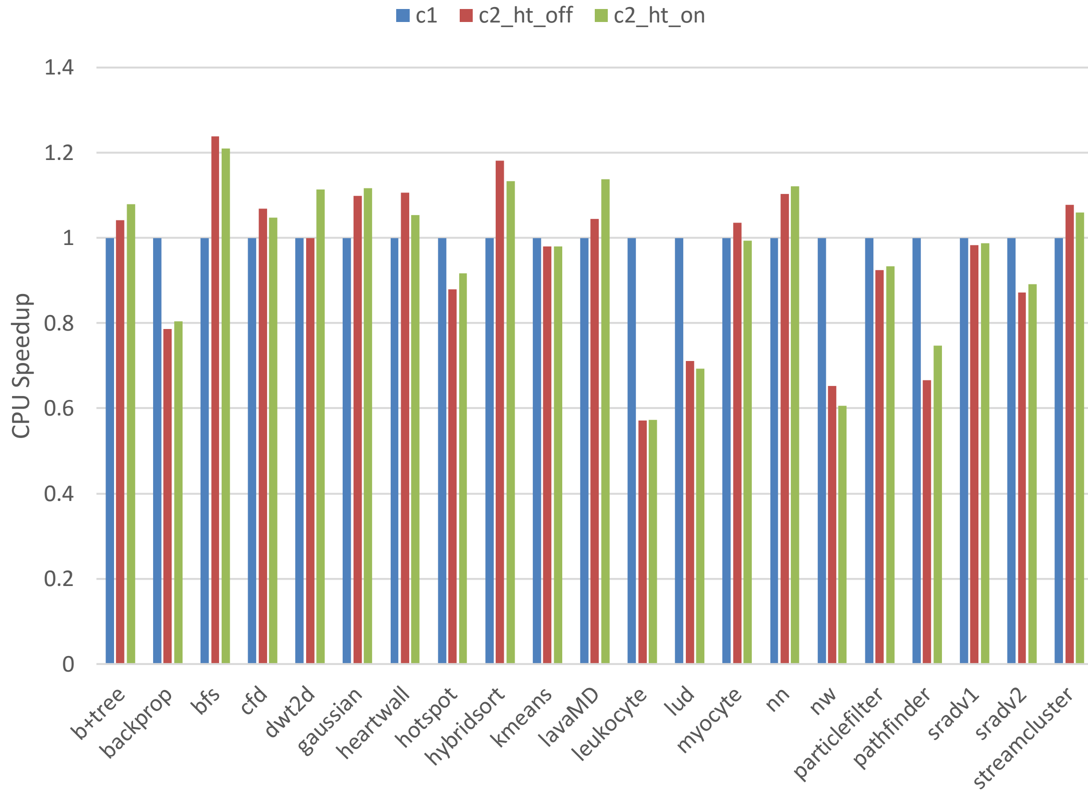


Figure 18: Scaling of CPU performance on *Intel Core i5-5200U* CPU.

From the experimental results, we see that execution times of the benchmarks on a CPU are dependent on its implementation and do not necessarily scale with the number of cores in a CPU. The vast difference in the observed GPU-normalized execution times is mainly due to serialization of blocks. Since different benchmarks have different number of blocks (which run in parallel in the GPU), and they are run sequentially by the simulator, the number of blocks in a particular benchmark also dictates its CPU performance. Furthermore, depending on the implementation of the application, some benchmarks were observed to have an increase in performance as the number of cores increased, while many benchmarks suffered a performance degradation. This may be

explained by additional synchronization and communication overhead negating the gains achieved by running in parallel on multiple cores.

We also see that the performance degradation in case of the *Intel Core i7-920* CPU compared to the *Intel Core i5-5200U* is much worse. A plausible explanation might be the more modern and optimized architecture, especially when it comes to the cache, memory and communication/synchronization/coherency architecture of the *Intel Core i5-5200U*, which is a 14nm processor based on Intel's Broadwell architecture. The *Intel Core i7-920* CPU design is a 45nm processor based on Intel's Nehalem architecture. But this needs further experimentation to point out the exact cause of the difference in performance.

Table 5 summarizes the execution times of all the benchmarks on an *Intel Core i7*-920 CPU for different configurations. Similarly, Table 6 summarizes the execution times of all the benchmarks on an *Intel Core i5-5200U* CPU for different configurations.

Also, for all the benchmarks, the basic block identification procedure was successfully completed. For the basic-block annotation procedure, CUDA 4.0 was used to generate the PTX. The PTX was used to identify the basic blocks, and then annotate the source code accordingly. After annotation, the annotated source code was passed through the simulator to generate the basic block execution sequence of individual threads. After post-processing the basic block execution sequence of each individual thread, the basic block execution sequence of each warp was successfully computed for all the benchmarks.

Benchmarks	CPU Execution Time (seconds)				
	c1	c2_ht_off	c2_ht_on	c4_ht_off	c4_ht_on
B+ Tree	0.777	0.989	0.990	0.896	0.961
Back propagation	11.361	20.434	21.658	19.917	22.477
BFS	29.295	46.110	49.864	46.803	40.475
CFD Solver	21.383	25.390	24.780	23.614	23.378
DWT2D	26.784	34.781	46.450	40.108	45.360
Gaussian	118.537	133.107	151.289	128.629	139.091
Heart Wall	4.633	5.649	4.953	4.811	4.864
HotSpot	13.094	22.209	23.633	22.028	23.707
Hybrid Sort	25.331	33.492	31.600	29.051	28.250
K-means	332.803	416.861	590.128	557.431	489.608
LavaMD	58.228	92.104	77.745	59.956	53.590
Leukocyte	133.657	349.151	449.650	359.651	444.906
LUD	0.285	0.604	0.728	0.641	0.733
Myocyte	4.920	8.008	8.320	8.022	7.407
Nearest-Neighbors	0.590	0.577	0.753	0.643	0.684
Needleman-Wunsch	15.123	38.469	41.578	40.388	40.433
Particle filter	1.208	1.615	1.877	1.741	1.870
Pathfinder	47.063	116.166	124.870	126.096	129.059
SRAD (v1)	2.595	3.031	3.481	3.443	3.468
SRAD (v2)	6.864	11.916	12.750	11.827	12.995
Stream Cluster	52.056	46.087	58.450	48.724	45.605

Table 5: Execution times of benchmarks for all configurations of *Intel Core i7-920*.

Benchmarks	CPU Execution Time (seconds)		
	c1	c2_ht_off	c2_ht_on
B+ Tree	0.763	0.733	0.707
Back propagation	8.956	11.405	11.142
BFS	26.083	21.066	21.554
CFD Solver	16.971	15.878	16.209
DWT2D	25.286	25.290	22.718
Gaussian	87.483	79.699	78.406
Heart Wall	4.785	4.327	4.540
HotSpot	10.020	11.393	10.931
Hybrid Sort	23.371	19.799	20.631
K-means	326.819	333.331	333.721
LavaMD	26.984	25.847	23.737
Leukocyte	154.257	269.948	269.493
LUD	0.306	0.431	0.442
Myocyte	3.733	3.604	3.757
Nearest-Neighbors	0.428	0.388	0.382
Needleman-Wunsch	14.560	22.327	24.034
Particle filter	1.210	1.308	1.297
Pathfinder	54.487	81.761	72.969
SRAD (v1)	2.552	2.597	2.587
SRAD (v2)	5.418	6.211	6.080
Stream Cluster	48.839	45.308	46.114

Table 6: Execution times of benchmarks for all configurations of *Intel Core i5-5200U*.

Chapter 6

Summary and Conclusions

The simCUDA project had two main aims – first, developing a functional simulator to execute GPU code on a multi-core CPU, and second, to model and estimate the sequence of execution of warps in a GPU. The functional simulator has been successfully developed such that it can run on any existing platform that supports C++11 threads. The simulator does not require any available GPU for simulation. Instead, it translates the CUDA application into functionally equivalent C++11 code, and it relies on C++11 threads to achieve parallelism through multi-core CPU architectures. Thus, simCUDA proves to be a useful tool for host-compiled simulation of GPU applications. Also, the simulator extension of extracting the basic-blocks works on an intermediate representation that is hardware agnostic. Thus, in essence almost no modification is required for it to be used on a variety of target platforms.

The project has been developed after a good understanding of the CUDA programming model, its requirements as well as its challenges. Also, a good knowledge of C++11 threads and their existing features was required to aptly model CUDA constructs of C++. Unfortunately, the documentation for the thread support library has been lacking in many aspects, which proved to be a major hurdle for overcoming issues and complications in the initial stages.

The simulator has been developed using Perl, and it uses complex regular expressions to make the necessary modifications. The project has been developed with the intent of keeping it as modular as possible so that it can be modified easily to suit the needs of the end-user as and when required.

The simulator has been successfully tested with the Rodinia benchmark suite, both as a functional simulator, and for generating the basic block execution sequences of warps. This was done to model real-world usage and applications. The execution times of the benchmarks differed depending on the number of cores, and the specific implementation of the application. For some applications, the performance scaled with the number of cores, while for some, performance degradation was observed.

It is possible to extend the simulator's functionalities in future depending on the requirements of a particular application. Also, it might be possible to extend the simulator to support different languages as well. Since the simulator relies on regular expressions to modify the source directly, supporting a new language is a possibility by simply adding new regular expressions. As another possibility, the underlying language can also be changed (e.g., to use C-Pthreads instead of C++11 threads). This is feasible, since it requires modifying the source code with a new equivalent construct.

In conclusion, this project has established itself as a viable concept for directly modifying source code in order to support mapping between different GPU and CPU architectures, and it provides a framework to support mapping between different architectures irrespective of the underlying hardware.

References

[1] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, et al. 'Implicitly Parallel Programming Models for Thousand-Core Microprocessors', *Proceedings of the 44th Annual Design Automation Conference,* San Diego, CA, USA, June 2007.

[2] Erik Lindholm, John Nickolls, Stuart Oberman, et al. 'NVIDIA Tesla: A Unified Graphics and Computing Architecture', *IEEE Micro,* vol. 28/no. 2, (2008), pp. 39-55.

[3] John Nickolls, Ian Buck, Michael Garland, et al. 'Scalable Parallel Programming with CUDA', *ACM Queue,* vol. 6/no. 2, (2008), pp. 40-53.

[4] Timothy Purcell, Ian Buck, William Mark, et al. 'Ray Tracing on Programmable Graphics Hardware', *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques,* San Antonio, TX, USA, July 2002.

[5] Juekuan Yang, Yujuan Wang, and Yunfei Chen. 'GPU Accelerated Molecular Dynamics Simulation of Thermal Conductivities', *Journal of Computational Physics,* vol. 221/no. 2, (2007), pp. 799-804.

[6] Ian Buck. 'GPU Computing with NVIDIA CUDA', *ACM SIGGRAPH 2007 Courses,* (2007), pp. 6-es.

[7] NVIDIA. ‘NVIDIA CUDA Compute Unified Device Architecture’, NVIDIA Corporation, Santa Clara, California, 2015. Web Resource. http://docs.nvidia.com/cuda

[8] IEEE. ‘IEEE std. 1003.1c-1995 thread extensions’, *IEEE 1995, ISBN 1-55937-375-x formerly posix.4a now included in 1003.1-1996.*

[9] MPI Forum. ‘MPI-2: Extensions to the Message-Passing Interface’, *University of Tennessee, Knoxville, TN, USA, Tech. Rep.*, July 1997.

[10] Rohit Chandra, Ramesh Menon, Leo Dagum, et al*., 'Parallel Programming in OpenMP'*, Burlington, Elsevier, 2000.

[11] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, et al. 'Analyzing CUDA Workloads using a Detailed GPU Simulator', *2009 IEEE International Symposium on Performance Analysis of Systems and Software,* Boston, Massachusetts, USA, April 2009.

[12] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, et al. 'Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems', *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques,* Vienna, Austria, September 2010.

[13] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. 'MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs', in *Languages and Compilers for Parallel Computing*, Springer Berlin Heidelberg, 2008, Volume 5335, pp. 16-30.

[14] Ian Buck. 'High Performance Computing with CUDA Tutorial: CUDA programmming environments ', *International Conference for High Performance Computing, Networking, Storage and Analysis,* Portland, Oregon, USA, November 2009. Web Resource. http://www.nvidia.com/object/SC09_Tutorial.html

[15] NVIDIA. 'CUDA Compiler Driver NVCC TRM-06721-001_v7.5 Reference Guide', NVIDIA Corporation, Santa Clara, California, 2015. Web Resource. http://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf

[16] Justin Luitjens, and Steven Rennich, 'CUDA Warps and Occupancy', in G*PU Computing Webinar, GPU Tech Conference, NVIDIA Corporation*, July 2011.

[17] Information Technology–Programming Languages–C++, ISO/IEC 14882:2011 2011

[18] Intel. 'First the tick, now the tock: Next generation Intel microarchitecture (nehalem)', Intel Corporation, Tech. Rep. 2008.

[19] Ron Cytron, Jeanne Ferrante, Barry Rosen, et al. 'Efficiently Computing Static Single Assignment Form and the Control Dependence Graph', *ACM Transactions on Programming Languages and Systems,* vol. 13/no. 4, (1991), pp. 451-490.

[20] Suhas Chakravarty, Zhuoran Zhao, and Andreas Gerstlauer. 'Automated, Retargetable Back-Annotation for Host Compiled Performance and Power Modeling', *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis,* Montreal, Canada, September 2013.

[21] Jayaprakash Pisharath, Ying Liu, Wei-keng Liao, et al. NU-MineBench 2.0. *Technical Report CUCIS-2005-08-01*, Department of Electrical and Computer Engineering, Northwestern University, Aug 2005.

[22] Michael Boyer, David Tarjan, Scott T. Acton, et al. 'Accelerating Leukocyte Tracking using CUDA: A case study in leveraging manycore coprocessors', *2009 IEEE International Symposium on Parallel & Distributed Processing,* Rome, Italy, May 2009.

[23] Vladimir Likic, 'The Needleman–Wunsch Algorithm for Sequence Alignment'. Web resource. http://scie-nce.marshall.edu/murraye/Clearer%20Matrix%20slide-%20show.pdf

[24] Matthew Goodrum, Michael Trotter, Alla Aksel, et al. 'Parallelization of Particle Filter Algorithms', *Proceedings of the 2010 International Conference on Computer Architecture,* Saint-Malo, France, June 2010.

[25] Shuai Che, Michael Boyer, Jiayuan Meng, et al. 'Rodinia: A Benchmark Suite for Heterogeneous Computing', *2009 IEEE International Symposium on Workload Characterization,* Austin, TX, USA, October 2009.

[26] Erik Sintorn, and Ulf Assarsson. 'Fast Parallel GPU-Sorting using a Hybrid Algorithm', *Journal of Parallel and Distributed Computing,* vol. 68/no. 10, (2008), pp. 1381-1388.

[27] Neural Networks for Face Recognition. Web resource. http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mit-chell/ftp/faces.html.

[28] Pawan Harish, and P. J. Narayanan. , 'Accelerating Large Graph Algorithms on the GPU using CUDA', in *Proceedings of 2007 International Conference on High Performance Computing*, Goa, India, December 2007.

[29] Jordan Fix, Andrew Wilkes, and Kevin Skadron. 'Accelerating Braided B+ Tree Searches on a GPU with CUDA'. *ACM Transactions on Database Systems*. 2009.

[30] Andrew Corrigan, Fernando F. Camelli, Rainald Löhner, et al. 'Running Unstructured grid-based CFD Solvers on Modern Graphics Hardware', *International Journal for Numerical Methods in Fluids,* vol. 66/no. 2, (2011), pp. 221-229.

[31] GPUDWT: Discrete Wavelet Transform accelerated on GPU. Web Resource. https://code.google.com/arch-ive/p/gpudwt/

[32] Lukasz G. Szafaryn, Kevin Skadron, and Jeffrey J. Saucerman. 'Experiences Accelerating MATLAB Systems Biology Applications.' *In Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits* 2009, in conjunction with the *36th IEEE/ACM International Symposium on Computer Architecture*, Austin, TX, USA, June 2009.

[33] Wei Huang, S. Ghosh, S. Velusamy, et al. 'HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design', *IEEE Transactions on very Large Scale Integration (VLSI) Systems,* vol. 14/no. 5, (2006), pp. 501-513.

[34] Christian Bienia,, Sanjeev Kumar, Jaswinder Singh, et al. 'The PARSEC Benchmark Suite: Characterization and Architectural Implications', *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques,* Toronto, Canada, October 2008.