A survey of verification tools for GPU software

THESIS

Submitted in partial fulfillment of the requirements of the course
BITS F421T

by

Anmol Panda

ID No. - 2012A7PS123G

Under the supervision of

Dr. Philipp Rümmer

Assistant Professor, Uppsala University

Dr. Neena Goveas

Associate Professor, BITS Pilani K K Birla Goa Campus

in the

Department of Computer Science and Information Systems

BITS Pilani K. K. Birla Goa Campus

2nd May, 2016

Declaration of Authorship

I, **Anmol Panda**, declare that this thesis titled, 'A survey of verification tools for GPU software' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for the first degree thesis at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

An ounce of practice is worth tonnes of preaching

Mahatma Gandhi

Abstract

A survey of verification tools for GPU software

by

Anmol Panda

Graphics Processing Units (GPUs) have been identified as highly suitable for data-intensive application in a wide range of domains such as image processing, super-computing, quantum physics and bio-informatics among others. GPU programming is the process of designing, writing and testing software that runs on GPUs. Due to the pervasive usage of GPUs and GPU programming, GPU software verification tools were developed to ensure their accuracy and reliability. In this thesis, we have considered two such tools: GPUVerify and GKLEE. Our objectives were to learn about the common challenges developers face in GPU programming, to understand the specific bugs that these two tools report and compare their scope and scalability aspects. We have also considered their usability features like the time of execution and the nature of their output and learn-ability issues like support and documentation. In order to test the software, twenty-six benchmarks were selected from open-source applications. These benchmarks were then verified using the tools and the results documented and analysed. These results suggest that GPUVerify is a useful lightweight tool to detect data races and barrier divergence. However, GKLEE subsumes GPUVerify by also reporting performance issues like bank conflicts, warp divergence and non-coalesced memory accesses. It therefore provides a holistic analysis of the accuracy and performance of a kernel. In contrast, he report concludes that GPUVerify is more portable and user-friendly than GKLEE and its output is easier to interpret. It should be used to test individual kernels in the initial stages of application development. On the other hand, GKLEE has a more detailed output and has greater system dependency than GPUVerify. It is best suited for testing whole applications than separate kernels. Lastly, we outline potential goals for future research.

Acknowledgements

I would like to begin by expressing my sincere gratitude to Dr. Philipp Rümmer, Assistant Professor, Department of Information Technology, Uppsala University for agreeing to mentor this project. His continued guidance has helped conceptualise the project's aim, define it's scope and work towards the expected outcomes of this thesis. The weekly online meetings have driven the research in the desired direction while ensuring the project meets its weekly deadlines.

I would like to extend my heartfelt gratitude to Dr. Neena Goveas, Associate Professor, Department of Computer Science and Information Systems, BITS Pilani K. K. Birla Goa Campus for agreeing to be the on-campus supervisor for this thesis. Given the unique set-up of the project, with the researcher in the campus and the mentor in Uppsala university, Dr. Goveas has been instrumental in getting the project streamlined to the university's requirements. It would not have been possible to undertake this work without her consent to be the supervisor.

I would also like to thank Dr. Bharat Deshpande, Head of Department, Department of CS and IS, BITS Pilani K. K. Birla Goa Campus for his guidance during the project. He has been helpful in providing necessary resources such as the computer system in lab A-200 for conducting vital experiments and and in granting permissions for the same.

I am also thankful to Dr. A. Baskar for agreeing to be the examiner for my thesis and giving me the opportunity to present my research on the eve of department day.

I am grateful to the Computer Center Lab management for providing the necessary technical infrastructure for the weekly Skype meetings with Dr. Rummer. Without their help and support, this communication would have been very difficult. Lastly, I am thankful to Mr. Anuj Khandelwal, my batchmate and friend for lending his laptop computer for these meetings.

Contents

List of Figures

List of Tables

Listings

Dedicated to my
Mamma

Chapter 1

Introduction

Given the rapid developments in multi-core processor technology, the use of general purpose GPUs has increased exponentially in the past decade. GPUs now play a vital role in various types of applications that rely on these chips for parallel computation. GPU computing platforms such as OpenCL, CUDA and OpenAMP have made a disruptive impact on the way data-intensive software are conceptualised and implemented. However, this dependence on GPUs raises important questions regarding accuracy and verifiability of such software. GPU kernels are prone to bugs such as data races that go undetected during manual debugging. Errors such as incorrectly placed barriers and inefficient memory accesses are difficult to find for humans.

Such errors, if left unchecked, can render the system in an undefined and unpredictable state [1], [2], [3]. In order to recover from such a state, a system reboot may be required. Since GPUs are now used in critical applications such as defence systems, aerospace systems and medical equipment, the possibility of a system crash cannot be tolerated.

Consequently, tools such as GPUVerify, GKLEE and PUG have been developed. These tools vary in their target applications, the scope of errors they detect in those applications and the approach they take to find and report them. They also vary in the depth of their testing. GPUVerify conducts a static analysis of CUDA and OpenCl GPU kernels and can report potential bugs, namely data races and barrier divergence. On the other hand, GKLEE operates only on CUDA kernels and reports not only the aforementioned bugs, but also analyses thread divergence within a warp, inefficient memory accesses and bank conflicts.

In this thesis, we have assessed the common errors that can occur in GPU software. We used these tools to test carefully selected OpenCL and CUDA benchmark kernels. The results of these tests have enabled us to comparatively analyse these tools not just for the accuracy of their claims, but also their robustness, versatility and usability aspects.

1.1 Scope and Objectives

The objectives of the thesis were to analyse these software for their similarities and differences as well as benefits and disadvantages of using each. We have compared the tools for the factors mentioned below and provide an overview of the advantages and drawbacks of each. We have also discussed major challenges facing GPU programming, with specific interest in bugs such as data races and barrier divergence.

The scope of the project was limited to the usage and benefits of deploying the said tools. We did not explore the mathematical and logical models of these tools. Similarly, the design and architecture of the software was also excluded from the analysis. We narrowed our focus to the types of bugs and performance issues, if any, that the tools report. Moreover, we also looked at factors such as system requirements, run-times and ease with which results can be interpreted by developers. Lastly, we considered the usability and learn-ability aspects of the tools as well.

In the subsequent chapter, we have described the various stages of this project. Firstly, chapter two explains the motivation and background of our research. Chapter three outlines the broad challenges in GPU computing paradigm. Chapter four documents the experimental setup and process while chapter five lists the results and analysis of those experiments. Finally, chapter six explains the conclusions of our research and the potential for future work in this area.

1.2 Related works

The development of massively parallel GPUs has enabled their usage in domains from personal computing, graphics applications, aerospace systems, medical imaging, etc. This has motivated researchers to address the need for verifying GPU software written for such myriad sectors. Betts et al [2] describe GPUverify, the background behind its need and development, the mathematical model used for verification and the performance of the tool. They also list certain drawbacks and possible improvements. Ethel Bardsley and Alastair Donaldson [4] explore the practical impact of design decisions, namely coarse-grained thread synchronization within the same warp on one hand and atomic operations on the other. Gudong Li and Ganesh Gopalakrishnan [3] describe the logical model of GKLEE, its capacity to detect bugs and performance issues and its performance during verification of select commercial SDKs. They also explain the architecture and test-generation model of GKLEE. Wei-Fan Chiang et al [5] describe a new method to detect bugs utilising both barriers and atomics. They present a new algorithm, test it on select benchmarks and provide the results. Guodong Li, Gopalkrishnan et al [6] describe the need for a GPU software verification tool and describe their model for a preliminary automated symbolic verifier called Prover of User GPUs (PUG). They utilise techniques like partial order reduction and loop abstraction in this tool. To the best of our knowledge, no studies of these tools similar to this one have been conducted. The papers mentioned above introduce, analyse and evaluate the said tools.

Chapter 2

Background

2.1 Emergence of Graphics Processing Unit

Until about 2003, rapid advances in performance of microprocessors was achieved through continuous increases in clock frequency, memory bandwidth and complexity of control logic. However, this process reached saturation due to limitations on energy consumption and heat dissipation. [7] The industry responded to this by generally using the available amount of transistors to realise parallel architectures, both on CPUs and the newly developed Graphics Processing Units (GPUs). This has enabled the earlier trend of performance growth and cost reduction to continue.

A Graphics Processing Unit is a dedicated multi-thread processor with thousands of cores that run multiple tasks in parallel. Its main purpose is to render images and motion picture but it is increasingly used to tackle other compute-intensive parts of software. By running the sequential code on the CPU and the compute-intensive portion on the GPU, massive gains in performance can be achieved.

The following table lists the important difference between CPUs and GPUs[7]

Winning applications use a combination of CPUs and GPUs to capture the advantages of both [7]. Domains that use GPU computing include financial analysis, scientific simulation, engineering simulation, data intensive analytics, medical imaging, digital audio and video processing, computer vision, biomedical informatics and electronic design automation [7]. Many of these domains consist of critical systems such as aircraft control and medical instruments that have a direct impact on safety. In such cases, the accuracy and predictability of software assume greater importance.

In this context, much work has been done on CPU software verification over the past few decades. Also, as CPU code is mostly sequential, error debugging and isolation is

Property	CPU	GPU
No. of Cores	Tens	Thousands
Core Type	Latency Oriented	Throughput Oriented
Local cache size	Large	Small
No. of registers	Relatively few	Relatively many
No. of SIMD units	Relatively few	Relatively many
Control logic	Sophisticated	Simple
Scheduling logic	Relatively simple	Relatively complicated

TABLE 2.1: Comparison between CPU and GPU

relatively easier. On the other hand, parallel programming using GPUs is a different process altogether. This creates new challenges in not just writing code but also debugging errors and verifying stated claims. Moreover, aspects such as scalability and portability requirements only add to the complexity.

2.2 GPU Programming Platforms

There are three main programming platforms that are commonly used for GPU software development [2]. These are OpenCL developed by the Khronos group [8], Compute Unified Device Architecture (CUDA) developed by Nvidia [9] and C++ AMP from Microsoft[10]. OpenCL is an open-source platform that is available freely and is AMD's main GPU programming model [2]. On the other hand, CUDA is a licensed platform developed by Nvidia for its GPUs [9], [11].

Let us now consider the programming model used in CUDA. It is significantly different from serial CPU programs and is illustrated here using the following example [7]. The sequential CPU code written in C without any parallel components of CUDA is listed in listing 2.1.

```
1 void vecAdd(float *h_a, float *h_b, float *h_c, int n){
2     int i;
3     for(i=0;i<=n;i++)
4         h_c[i] = h_a[i] + h_b[i];
5 }
```

LISTING 2.1: Sequential Vector Addition on CPU

This code executes only on the CPU and has no interaction whatsoever with the GPU.

On the other hand, the parallel CUDA program has two distinct components: Host code and Device code.

The Host code is sequential and runs as before on the CPU (Host). It code initialises the data items i.e. vectors **A**[ ] and **B**[ ] and calls the CUDA kernel. In the CUDA platform, the Host calls a kernel function to run a multi-threaded section of the code on the GPU (Device).

The kernel function accepts pointers to the data items as parameters. It also takes two configuration parameters, namely number of blocks and threads per block. On the GPU, each thread runs the same kernel code for a different data item. The kernel code stores the sum in vector **C**[ ] and returns a pointer to **C**[ ] to the Host. The Host then accepts the pointer and copies the data in C to its local data item. Listings 2.2 and 2.3 below illustrate Host and Kernel code respectively.

```
1 #include <stdio.h>
2 #include <cuda.h>
3 void vecAdd( float* h_a, float* h_b, float* h_c, int n){
4     int size = n*sizeof*float);
5     float* d_a, d_b, d_c;
6     //allocate memory for Vectors
7     cudaMalloc((void**) &d_a, size);
8     cudaMemcpy( d_a, h_a, size, cudaMemcpyHostToDevice);
9     cudaMalloc((void**) &d_b, size);
10    cudaMemcpy( d_b, h_b, size, cudaMemcpyHostToDevice);
11    //call kernel function with configuration parameters
12    vecAddKernel<<<ceil(n/256.0), 256>>>( d_a, d_b, d_c, n);
13    //copy vector C to host memory
14    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
15    cudaFree(d_a);
16    cudaFree(d_b);
17    cudaFree(d_c);
18 }
```

Listing 2.2: Cuda Host code

```
1 __global__
2 void vecAddKernel( float* A, float* B, float* C, int n){
3     int i = threadidx.x + blockDim.x*blockIdx.x;
4     if( i < n )
5         C[i] = A[i] + B[i];
6 }
```

Listing 2.3: Cuda Kernel code

We can now consider the program in the examples above. As mentioned earlier, it consists of two entities: Host and Device. The **Host** refers to the **CPU** and the **Device** refers to the **GPU**. The application runs on the Host. Thus all sequential parts of the

code run on the Host as before. But the Host transfers control to the Device to run the parallel sections of the code. This transfer of control is achieved through **kernel** functions. They are similar to functions in C but also accept two configuration variables as parameters, namely the number of blocks and threads per block respectively. These variables determine the dimensions of the grid and blocks respectively. Apart from these, the kernel accepts normal parameter variables like pointers to arrays of data items. Each thread runs the same kernel code on a separate data item. Assuming that the number of threads is more than the number of data items, all data items are processed in one step.

Using the cudaMalloc() function, memory space can be reserved for data items. The cudaMemcpy() function can transfer data between the Host and Device. In the example above, the data array *h_a* is copied into array *d_a* which is then passed to the kernel function as data. The variable size refers to the total memory in bytes used by each array. The variables **cudaMemcpyHostToDevice** and **cudaMemcpyDeviceToHost** are predefined in cuda.h and indicate the direction of transfer i.e from CPU to GPU or vice-versa, respectively.

The **__global__** implies that the following function can be called by the host but is executed on the device. Thus, it is used before the kernel function **vecAddKernel()**. The parameters *A, B and C* are each a pointer to an array of data items. Vectors A and B refer to the input arrays and vector C stores the resulting sum. Integer n stores the number of data items. Inside the function, integer variable i stores the id of each thread, calculated using system variables threadIdx.x, blockDim.x and blockIdx.x. The variable threadIdx.x refers to the local id of a thread, within its block. The **.x** reference is used to refer to the x co-ordinate of the id, as blocks may have one, two or three dimensions. In the case of a 2 D block, both x and y co-ordinates must be used to refer to an individual thread. Variable blockIdx.x refers to the x co-ordinate of the block id while blockDim.x refers to the x dimensions of the grid of blocks. Similarly, blockDim.y will refer to the y dimensions. These variables are defined in the header file cuda.h. The condition $i < n$ ensures that only those threads that have a legitimate data item to operate upon are executed. This prevents any accesses to memory beyond the scope of the program. The function $\lceil n/256.0 \rceil$ ensures that there are adequate number of threads in the device to run all data items. The number 256 here refers to the number of threads per block. Thus the grid on the device will have exactly $\lceil n/256.0 \rceil$ blocks. Each of these threads runs one iteration of the kernel function on one unique data item. The array C is then copied back to the Host using **cudaMemcpy()** function and stored in data array *h_c*. Lastly, the Host releases the memory used by local variables *d_a, d_b* and *d_c* using the **cudaFree()** function.

2.3 Verification of GPU software

One can observe the significant differences between sequential programming and parallel CUDA programming. It is therefore essential that we take a closer look at what new challenges emerge from this model. For example, to achieve maximum throughput, it is essential to keep all fine grained CPU threads busy[6]. Moreover, it is necessary to ensure coalesced data movement (combining several memory accesses together to reduce data transfer overheads) between the global memory, used by the CPU and GPU, and the shared memory accessed by the GPU threads [6]. Also, bank conflicts need to be reduced and avoided as far as possible [6] to reduce data latency. The most common issues in GPU kernel programming are data races and barrier divergence [2]. The emulator that comes with standard GPUs does not cover all possible bugs and schedules, thus leaving the possibility of bugs escaping their scrutiny. Such bugs can cause GPU hardware to crash or deadlock [6]. More importantly, bugs such as data races occur only in specific scenarios and can therefore remain latent in code for a long time. But on transferring to a faster GPU, the same bugs can cause the program to stop working. It was therefore considered essential to develop GPU kernel verification software to automatically detect commonly found bugs in the code. Researchers have responded to this need over the past decade and several tools are now available to verify whether a kernel is free of bugs. We have selected three such tools in this thesis.

Firstly, GPUVerify [2] can be used to detect data races and barrier divergence in GPU kernels written in CUDA and OpenCL. It uses automatic abstraction techniques to generate conditions that are verified by automated theorem proving. It is used widely for verifying GPU kernels. Secondly, GKLEE [3] can be used to verify GPU kernels written in C++. It can detect bugs such as data races and barrier divergence. It can also report non-coalesced memory accesses, memory bank conflicts and divergent warps (warps are groups of threads within a block; these threads are executed in parallel) [2] [3]. It is based on testing a program using concrete and symbolic (concolic) execution [3]. Thirdly, Prover of User GPU Programs (PUG) [6] is a kernel verification tool for kernels written in CUDA C. It takes a kernel written in C as an input, analyzes it using SMT tools and detects bugs such as data races, bank conflicts, incorrectly placed barriers,etc.

In this project, we have tested these software for the bugs they detect, run-times for different kernels, hardware requirements and usability issues among others. In the first phase of the project, benchmarks kernels were chosen from existing open-sourced libraries. In the second phase, these kernels were verified for bugs using the above mentioned software. We also wrote some simple CUDA kernels on our own to look for specific cases of bugs. In the last phase, we have completed a comparative analysis of these software.

Chapter 3

Challenges in GPU Computing

As we have seen in chapter 2, general purpose GPUs (GPGPUs) now play an ever increasing role in data-intensive applications. However, several major challenges emerge in designing and programming software that can fully harness the computing power of these GPUs. In this chapter, we list the major issues and isolate the specific programming bugs that we have considered in this project.

3.1 Differences in the programming model

Firstly, the GPU programming model is parallel in nature. It requires the user to understand parallel execution of programs and how to write code for the same. Since most developers learn programming through sequential languages like C, Java, Python, etc., their skills are tuned to designing applications that work well on CPUs. Consequently, it takes significant effort to learn parallel programming concepts and use them correctly when developing software for GPGPUs. Moreover, it is not easy to convert any given sequential program to a parallel version. Often, due to inefficient use of parallel APIs, the parallel code running on a GPGPU may perform worse than the sequential program running on a CPU [12] [13].

Secondly, the continuous rapid advances in GPU hardware technology have forced developers to change the programming platforms at regular intervals. Unlike CPUs, that maintain API compatibility across several generations, APIs written for older generation of GPUs may be sub-optimal and at times, even unusable on a new generation GPU [14]. At the same time, there are no definitive trends about the advances being made in GPU hardware, thus leaving few possibilities for programmers to plan ahead. Consequently, GPGPU programmers are faced with a dual problem: lack of backward compatibility for

the existing code and inability to anticipate future changes [14]. This adds significant uncertainty in developing applications for GPGPUs.

Thirdly, not all applications can be ported to a GPU. Several important problems do not have massively parallel algorithms [15]. Given the degree of parallelism needed to fully harness the potential of a GPGPU, many algorithms that are considered scalable may fall short of this standard. Some problems such as finding the shortest path and Delaunay triangulation do not have massively parallel algorithms that are work-efficient i.e. we cannot make them parallel without making the code much more complex or not at all. Also, applications that are highly parallelizable may not remain numerically stable, like the Parallel Cyclic Reduction (PCR) algorithm [15].

3.2 Differences in GPU and CPU architecture

Firstly, programmers must adapt to the architectural changes between CPUs and GPUs. For example, a CPU, with tens of cores, has sufficient cache memory available to speed up data access. On the other hand, a GPU with thousands of cores has a shared on-chip memory that is used as a cache. Here the programmer must rely on the scheduler to appropriately use the on-chip memory and reduce data transfers from the global memory [12]. Inefficient usage of the shared memory can lead to recurring transfers to and from the larger global memory, thus adding another major overhead. Also, memory accesses from different threads need to be properly coalesced so as to reduce data transfer latency [16]. Another aspect of these architectural constraints is the degree of multi-threading that the programmer must deploy in the application. Using very few threads under-utilises the GPUs resources. The program must therefore use thousands of threads to adequately exploit the GPUs data crunching power and prevent it from under-performing. However, if too many threads are used, the cores may run out of registers, thus forcing the GPU to simulate additional registers from its on-chip memory, further increasing data latency.

Secondly, while GPU programming has emerged as a major implementation of parallel computing, there are several other architectures available in the industry such as multi-core CPUs and stream processors. [14]. For GPU computing to be accepted as the default form of parallel computing it must establish application portability across all of these major domains. While this may not appear to be absolutely necessary today, developers can save significant time and resources if programs written for GPUs can run efficiently on other architectures and vice-versa [14]. It will also enable GPUs to benefit from advances in the future in other domains of parallel computing.

3.3 Optimizing performance

Firstly, programming on GPUs imposes several constraints compared to serial code written for CPUs. For, example CPUs have sophisticated control logic with branch and loop prediction that optimize execution. Contrary to this, the GPU has very little support for branching and looping which leads to inefficient execution. Also, the slowest part of the process is to move the data in and out of the GPU [17]. Thus, it is essential to profile GPGPU applications, quantify the overhead due to these copy operations and determine if it acts as a performance bottleneck. Appropriate changes in the design of the program must be made to remove or minimise such delays.

Secondly, optimizing performance for a GPU is a painstakingly difficult process, with little theoretical basis. While CPUs benefit from advanced profilers and optimizing compilers, such tools for GPU programs remain primitive. The few libraries that do exist for optimizing performance are too numerous and can add complexity to the code [12], making it less modular and difficult to debug. Using too many libraries can also add several dependency constraints to the application, reducing its portability. Moreover, there are few mechanisms available to provide feedback to the programmer about the performance of the GPGPU. For instance, minor changes in the code like use of one API function call instead of another can lead to drastic increase or decrease in performance, referred to as a performance cliff [14]. It is important that the system provides appropriate feedback to the user so as to set standards for efficient usage of the GPU resources and avoid such random spikes or falls in performance. Tools such as GPUVerify and GKLEE are a move in this direction.

3.4 Degree of manual intervention

In the current era, software that run on CPUs requires very little manual intervention for error detection, isolation and avoidance. However, such tools for GPGPU software remain scarce and limited in scope. Firstly, the CUDA framework imposes a great deal of manual control over variable placement as data transfer across different levels of the memory is not automated in the compiler [18]. Not only does this add to the workload of the programmer, it significantly raises the possibility that inefficient memory accesses remain uncorrected in the application. Consequently, many applications may only achieve about half the decent performance expected from a CUDA application and few would exceed 10 percent of their peak performance [18].

Secondly, while chip manufacturers claim that tremendous increases in performance can be achieved by using the GPGPU for parallel applications, such improvements can

only be achieved by the most motivated programmers willing to invest significant time, effort and resources in the often redundant tasks of manually optimising the code [19]. Moreover, such investments can be made only by a few sectors of research like the defence sector or the financial services industry, both of which tend to have a unique tolerance for persistent, albeit expensive and tiring, efforts to achieve desired results [18]. On the other hand, novice programmers like undergraduate students, young researchers and small start-ups have neither the motivation nor the resources for similar endeavours. this greatly reduces the degree of adoption of platforms such as OpenCL or CUDA [20] [19].

3.5 Other aspects

Apart form the programming aspects of GPGPUs, there are several other issues that may affect the programmers decision to switch to GPUs for parallel computations. Firstly, not all applications require the degree of speedup that a GPU can deliver. In many cases, other features in the architecture may act as bottlenecks, thus denting the performance benefits achieved by the GPU. Secondly, developing applications for platforms such as android is made much easier due to the availability of IDEs such as Android Studio and Eclipse. Similarly, programming platforms such as Visual Studio attract many more young developers due to their easy learn-ability features. While GPU programming platforms like OpenCL and CUDA do not directly compete with these platforms, they lose out a large proportion of potential developers, being relatively difficult to learn and use. Lastly, applications that make use of GPUs like high-definition games or movies, supercomputers, data centers, aerospace or defence applications require persistent efforts over a significant time period to complete. On the other hand, domains such as application development for handheld devices and web development motivate many more developers due to their simple usability features. Results in such applications are comparatively easier to achieve and market to potential users. They can also be altered more frequently with little effort. These factors adversely impact the popularity of GPU programming as choice for research, especially among students.

3.6 Common bugs in GPU kernels

While the issues mentioned above broadly cover the current challenges faced by programmers in the GPU computing domain, we have narrowed our focus to the prevalence of bugs in GPU kernel code. In the course of this thesis, we have considered certain programming and some performance bugs that can occur in GPU software. These include data races and incorrectly placed barriers. Let us consider them in greater detail.

3.6.1 Data Race

A data race is a common bug found in GPU kernels. It occurs when two or more threads try to simultaneously access data from the same memory location. If both the accesses are reads, the races is harmless but even if one of the threads is writing to the memory, the outcome of the two memory accesses is undefined. Consequently, the state of the application and by extension, that of the system is unpredictable.

```
1 #include<stdio.h>
2 #include<cuda.h>
3 __global__
4 void addToNextKernel(int *a, int b, int n){
5     int i = threadIdx.x + blockDim.x*blockIdx.x;
6     if( i < n )
7         a[i+1] = a[i] + b;
8 }
```

Listing 3.1: Data Race example

Listing 4 is a common example of a data race [6]. The kernel accepts an integer array **a**[ ], an integer **b** and integer n, where **n** is the number of elements in **a**[ ]. The kernel then adds **b** to each element in **a**[ ] and stores it in the next element of **a**[ ]. Consider threads with $i = 1$ and $i = 2$. Let us label them τ_1 and τ_2 respectively. As we know, all threads in a given block execute the kernel code in parallel. When τ_1 runs, it accesses two data items of array **a**[ ], i.e it reads **a[1]** and writes to **a[2]**. Similarly, when τ_2 runs it reads from **a[2]** and writes to **a[3]**. Therefore, both threads access the same data location **a[2]** and one of them is writing to it. This condition is referred to as a data race as both threads compete for the same memory location and the outcome is unpredictable since one of them is a **write**. In the example above, all threads τ_1 to τ_{n-1} have data races.

Some of these data races may be actual, others could be benign while some occur only for certain values of configuration parameters.

3.6.2 Barrier Divergence

The next example that we consider is barrier divergence. Barriers are introduced to synchronise threads to prevent bugs such as data races. In CUDA, the **__syncthreads()** function call is used to synchronise threads. In OpenCL, the function **barrier()** is used for the same purpose. However, often these barriers are placed incorrectly, thus leaving open the possibility that some thread skips the barrier. One such example is listed below. The kernel takes an array of integers as input and adds the even integers and subtracts all the odd ones. It returns the resulting sum in the variable **sum**.

```
1  __kernel void addEvenSubtractOdd(__global int *numbers, int n, __global int
       *sum){
2
3          int res=0;
4          int i, temp;
5          i = get_local_id(0);
6          if(i<n){
7                  temp = numbers[i];
8                  if(temp%2 == 0){
9                          res += temp;
10                         barrier(CLK_GLOBAL_MEM_FENCE);
11                 }else{
12                         res -= temp;
13                         barrier(CLK_GLOBAL_MEM_FENCE);
14                 }
15         }
16         *sum = res;
17 }
```

Listing 3.2: OpenCL example of Barrier Divergence and Data Race

Another common instance of barrier divergence occurs when barriers are within the scope of conditional statements. The barrier is well synchronised if the condition evaluates to the same value for all thread that execute parallel. However, even if one thread evaluates the conditional to some other value, the output is undefined with the possibility of unintended side-effects [1]. In such a case, if some threads skip the scope of the condition while others wait at the barrier inside the scope, the threads that have skipped the barrier will never encounter it. Consequently, the waiting thread will never be released, thus leaving the system in an unstable and unpredictable state. Thus, it can be noted that while barriers are used to synchronize threads across different parallel computing domains, the problem becomes particularly acute for GPU programming due to the reasons mentioned above.

[1] Section 12.4, "Synchronising Divergent Threads in a Group", CUDA Toolit Documentation [1]

Sr. No.	Verification Software	Programming Bugs		Performance issues		
		Data Race	Barrier Divergence	BCR	WDR	MCR
1	GPUVerify	✓	✓	×	×	×
2	GKLEE	✓	✓	✓	✓	✓

TABLE 3.1: Bugs and issues reported by GPUVerify and GKLEE

Among the tools we have chosen for analysis, GPUVerify reports both data races and barrier divergences. GKLEE, on the other hand, reports not just these bugs, it also does a runtime analysis of the kernel and locates any bank conflicts, warp divergences and non-coalesced memory accesses.

Chapter 4

Experiments and Testing

In this chapter, we outline the procedure followed when conducting experiments with the verification tools. It details the process of selecting benchmarks kernels, the specifications of the computer system used for testing and the system pre-requisites for both software. It also includes a sample run of two different test kernels with GPUVerify and GKLEE.

4.1 Choice of Benchmarks

The tools were to be tested on a wide range of simple kernels that would be easy to interpret. Therefore, the benchmarks were taken from among open source projects available on GitHub. The applications that have been chosen cover a wide variety of domains such as image processing, mining tools, mathematical operations, etc. These include kernels used for mathematical computation such as vector addition, matrix multiplication, estimation of PI and MonteCarlo functions. Some benchmarks were chosen from the list of examples provided by the developers of GKLEE. This enabled us to understand the behaviour of GKLEE in greater detail. In all, twenty-six kernels were selected for testing.

Secondly, the dependencies of the programs and their compatibility with the system affected the choice of benchmarks. Several applications had to be rejected as they were incompatible with the system or were dependent on libraries that could not be installed due to compatibility issues. Moreover, the different approaches adopted by GPUVerify and GKLEE also played a major role in the selection of kernels for benchmarks. While GPUVerify can easily test a kernel without any **main** function or header files, GKLEE requires a main function that calls the kernel. In order to test the entire application

using GKLEE, all dependencies such as header files and third-party libraries are required. Such variations have forced us to search extensively for kernels that are compatible with both the tools, thus delaying the process to a certain extent. It also reduced considerably the number of kernels that were tested using both tools.

4.2 Selected Benchmark Kernels

Id	Kernel	Programming Platform	Length of Code	Tested using
1	Transpose kernel [21]	OpenCL	78	GPUVerify
2	Matrix Mul [21]	OpenCl	76	GPUVerify
3	MatrixVectorMul [21]	OpenCL	57	GPUVerify
4	Loop4 [22] A.1	OpenCL	26	GPUVerify
5	Harlan-nested-kernel [22]	OpenCL	20	GPUVerify
6	NBody Computation A.2 [23]	CUDA	64	GPUVerify
7	PI_Estimation [23]	CUDA	74	GPUVerify
8	MatrixMultiply [24]	CUDA	101	GPUVerify
9	ImageBlur [25]	CUDA	178	GPUVerify
10	Pairwise sums timed [26]	CUDA	112	GKLEE & GPUVerify
11	GPU kmeans [26]	CUDA	224	GKLEE & GPUVerify
12	Vectorsums [26]	CUDA	47	GKLEE & GPUVerify
13	Matmul [26]	CUDA	112	GKLEE & GPUVerify
14	Pairwise sums [26]	CUDA	77	GKLEE & GPUVerify
15	Cube [27] A.6	CUDA	40	GKLEE & GPUVerify
16	Square [27]	CUDA	47	GKLEE & GPUVerify
17	Deadlock0 [28]	CUDA	49	GKLEE
18	Deadlock2 [28]	CUDA	66	GKLEE
19	Seive1 [28]	CUDA	86	GKLEE
20	Simple Error handling [25]	CUDA	187	GKLEE
21	Inter Block Data Race [28]	CUDA	15	GKLEE
22	Memory [29]	CUDA	66	GKLEE
23	Bank Conflict [28]	CUDA	49	GKLEE
24	Loop4a [22] A.3	OpenCL	27	GPUVerify
25	SumMatrix 2D grid 2D block [30]	CUDA	132	GKLEE
26	SumMatrix 1D grid 2D block [30]	CUDA	137	GKLEE

TABLE 4.1: Benchmarks selected for testing

Table 4.1 lists the benchmarks selected for testing, the programming platform they belong to, the length of their code and also the verification tools that were used in the experiments. These kernels have been chosen after a thorough search of GitHub repositories and code samples. They represent common usage of GPUs like matrix operations and parallel sums. **Transpose kernel**, **Matrix Mul**, **MatrixMultiply**, **MatrixVectorMul** and **Matmul** are kernels that deal with operations on matrices. On the other hand, **loop4** and **loop4a** are nested for loops. **Vectorsums**, **Pairwise sums** and **Pairwise sums timed** conduct different addition operations on a given set of integers. **PI Estimation** estimates the value of π while **N-Body computation**, as the name suggests, conducts an n-body computation. Cube and Square calculate cubes and squares of integers respectively.

The benchmarks **Deadlock0**, **Deadlock2**, **Seive1**, **Inter Block Data Race** and **Seive1** were chosen from the same GitHub repository as that of GKLEE [28] and are provided by the developers. These act as illustrative examples of the way GKLEE reports potential deadlocks due to barrier divergence and bank conflicts.

4.3 Experimental Setup

The benchmarks were then tested for bugs using the chosen verification tools, namely GPUVerify and GKLEE. The experiments were conducted on a computer in Lab A-200 of BITS Pilani K. K. Birla Goa Campus, India. The system details are listed in table 4.2.

Sr No	Property	Type / Value
1	CPU	Intel ®Core ™ i7-3770
2	Clock Speed	3.40 GHz
3	Number of Cores	8
4	Graphics	Intel ®IvyBridge Desktop
5	Operating System	Ubuntu 14.04 LTS
6	OS Type	64 bit
7	System Memory	8 GB
8	Disk Size	483.8 GB

TABLE 4.2: System Specifications

4.3.1 GPUVerify Prerequisites

The GPUVerify Documentation [31] details the procedure for installing the tool. The software can be installed in two separate ways - Docker installation and Nightly builds. Docker is used to package the entire application into a portable and reproducible environment. However, the nightly builds allow us to directly install GPUVerify from source on both Linux as well as Windows platforms. The following are the prerequisites for installing and running GPUVerify on system 4.2.

Sr No	Prerequisite Tool	Remarks
1	Python [32]	Python programming platform version 2.7 or above
2	pip [33]	Tool for installing Python software packages
3	psutil [34]	Python module for cross-platform process and system utilities
5	Mono [35]	A software platform to create cross platform applications on the .NET framework
5	gcc [36]	GNU Compiler Collection
6	GPUVerify	Linux compatible nightly build of GPUVerify

TABLE 4.3: Prerequisites for GPUVerify on system 4.2

4.3.2 GKLEE prerequisites

The Github page of GKLEE consists of the entire source code along with the installation and usage instructions. GKLEE can be downloaded from its repository and installed, subject to the following prerequisites on system 4.2.

Sr No	Prerequisite Tool	Remarks
1	flex[37]	Lexical Analyzer that generates scanners
2	bison [38]	General purpose parser generator that converts an annotated CFG to a deterministic LR parser
3	CMake [39]	Open-source build tool (Version 3.0 or above)
4	git[40]	Open-source version control system for software

TABLE 4.4: Prerequisites of GKLEE on system 4.2

4.4 Prover of User GPUs

Prover of User GPUs (PUG) was the third tool chosen for analysis in this project. However, the latest vesion of the tool has been ported to GKLEE [41] by its developers i.e. PUG has now been subsumed by GKLEE. Hence, it was decided to restrict the scope of the project to GPUVerify and GKLEE. The time constraints of the thesis were also a major factor in this decision.

The tools, GPUVerify and GKLEE, have been installed and test run on the system. The benchmark kernels listed in table 4.1 have been tested one at a time using these tools. A sample test run of both software is explained in the following section.

4.5 Analysing kernels using GPUVerify and GKLEE

GPUVerify and GKLEE are different software by nature: GPUVerify is a purely static verification tool, GKLEE is a concrete and symbolic (concolic) analyser of GPU kernels. The aim and scope of both the tools are different as well, although there is some overlap. Consequently, the testing requirements, their input files, test procedure, run-times and output differ significantly.

4.5.1 GPUVerify

On downloading the GPUVerify nightly build for Linux and extracting it, it is seen that the directory has only one folder named for the date when the latest version was officially released. Inside this folder, all the files required to run the tool are present. To test run the system, we must first write a sample kernel. Some samples are provided in the package, but we have used a new kernel 3.2.

Figure 4.1 describes the output of GPUVerify. The command
./gpuverify –local_size=2 –num_groups=1 barrierDivergence.cl
calls the GPUVerify script and passes the OpenCL kernel file as input to it. The variables **local_size** and **num_groups** refer to the number of work items per group (threads per block) and number of work groups in the grid (number of blocks) respectively. In this case, the number of work items per group is 2 and the number of groups is 1. As can be observed, GPUVerify detects two different types of errors. Firstly, there is a possibility of divergence in the barriers in line 10 and line 13 of the code and each of these is listed as a separate error. Secondly, there is a chance of a data race between two work items (threads) when writing to variable **sum** in line 16 as both thread access the same data item.

```
root@anmol-OptiPlex-7010:/home/anmol/Thesis/GPUVerify/2016-02-16# ./gpuverif
y --local_size=2 --num_groups=1 ../../Example_Code/barrierDivergence.cl
barrierDivergence.opt.bc: warning: Assuming the arguments 'numbers', 'sum' o
f 'addEvenSubtractOdd' on line 1 of ../../Example_Code/barrierDivergence.cl
to be non-aliased; please consider adding a restrict qualifier to these argu
ments

../../Example_Code/barrierDivergence.cl:10:25: error: barrier may be reached
 by non-uniform control flow
  barrier(CLK_GLOBAL_MEM_FENCE);

Bitwise values of parameters of 'addEvenSubtractOdd':
  n = 3


../../Example_Code/barrierDivergence.cl:13:25: error: barrier may be reached
 by non-uniform control flow
  barrier(CLK_GLOBAL_MEM_FENCE);

Bitwise values of parameters of 'addEvenSubtractOdd':
  n = 1


../../Example_Code/barrierDivergence.cl: error: possible write-write race on
 sum[0]:

Write by work item 0 with local id 0 in work group 0, ../../Example_Code/bar
rierDivergence.cl:16:14:
  *sum = res;

Write by work item 1 with local id 1 in work group 0, ../../Example_Code/bar
rierDivergence.cl:16:14:
  *sum = res;

Bitwise values of parameters of 'addEvenSubtractOdd':
  n = 3

GPUVerify kernel analyser finished with 0 verified, 3 errors
root@anmol-OptiPlex-7010:/home/anmol/Thesis/GPUVerify/2016-02-16#
```

FIGURE 4.1: Sample run of GPUVerify on kernel 3.2

4.5.2 GKLEE

The GKLEE webage [42] links to the official Github repository of GKLEE [43]. The README file in the repository includes all the information required to install and run the software. The command

gklee-nvcc inter_block_race.cu

compiles the source code using Nvidia's NVCC compiler and creates an executable **inter_block_race**. Then, the command

gklee inter_block_race

must be run to exectute GKLEE. GKLEE runs the program and checks for various errors such as different types of data races, potential deadlocks due to differing paths taken by threads, rate of bank conflicts, rate of memory coalescing and rate of Warp

```
********** Start checking races at Device Memory **********

[GKLEE]: Under the pure canonical schedule, across different blocks, thread
0 and 64 incur a Write-Write race (Actual) on
[GKLEE] Inst:
Instruction Line: 6, In File: inter_block_race.cpp, With Dir Path: /home/anm
ol/Thesis
[File: /home/anmol/Thesis/inter_block_race.cpp, Line: 6, Inst:   in[threadId
x.x] = blockIdx.x;]
  store i32 %2, i32* %6, align 4, !dbg !715
<W: 44906512, 0:0, b0, t0>
[GKLEE] Inst:
Instruction Line: 6, In File: inter_block_race.cpp, With Dir Path: /home/anm
ol/Thesis
[File: /home/anmol/Thesis/inter_block_race.cpp, Line: 6, Inst:   in[threadId
x.x] = blockIdx.x;]
  store i32 %2, i32* %6, align 4, !dbg !715
<W: 44906512, 0:1, b1, t64>
[GKLEE]: One thread at BI 1 of Block 0 incurs a write-write race with the th
read at BI 1 of Block 1
KLEE: ERROR: /home/anmol/Thesis/inter_block_race.cpp:16: execution halts on
encounering a (global) race
```

FIGURE 4.2: Excerpt from a sample run of GKLEE on kernel A.4

Divergence. Figure 4.2 illustrates an excerpt of a sample run of GKLEE on the kernel A.4.

The selected benchmarks were then tested using GPUVerify and GKLEE. The following chapter documents the results and their analysis.

Chapter 5

Results and analysis

In this chapter, we detail results of the tests that were conducted. We have documented the number of errors and their types that were reported during the tests. The runtimes are also listed. In sub-section 5.3, a comparative analysis of the two software and their performance has been included.

5.1 Verification using GPUVerify

Testing with GPUVerify was a simple process compared to GKLEE. For each benchmark, the kernel functions(s) were extracted and stored in a separate file. This file was then passed as an input to the GPUVerify script along with the configuration parameters. For CUDA kernels, the parameters are called **blockDim**, refering to the dimensions of an individual block and **gridDim**, which refers to the dimensions of the grid. These parameters were set to single dimension values **blockDim = 16** and **gridDim = 16**.

GPUVerify takes relatively less time to complete its analysis. The runtimes observed in the tests vary between hundreds of milliseconds to 40 seconds, but never above one minute. The output of the program is relatively concise and easier to interpret and analyse. Moreover, GPUVerify detects two separate types of potential bugs, namely data races and barrier divergence. We have documented the results in the following tables.

In tables 5.1 and 5.2, the number of instances of both errors, data races and barrier divergence, are listed. In each of the cases, except N-Body Computation, GPUVerify completes its execution in less than ten seconds. Moreover, GPUVerify reports each

[1] GPUVerify exits with error: unhandled exception

Id 4.1	Benchmark	Data Race #	Barrier Divergence #	Time
1	Transpose kernel	4	0	1.7s
2	Matrix Mul	2	0	1.8s
3	Matix vector Multiply	0	0	1.6s
4	Harlan Nested Kernels	5	0	2.5s
5	Loop4	1	0	3.4s
24	Loop4a	Err [1]	Err	17.8s

TABLE 5.1: Results: OpenCL benchmarks verified using GPUVerify

Id 4.1	Benchmark	Data Race	Barrier Divergence	Time
6	N-Body Computation	2	2	39.7
7	PI Estimation	3	0	3.9
8	MatrixMultiply2	8	0	6.7
9	Image Blur	0	0	0.7
10	Pairwise sums timed	4	0	1.6
11	GPU kmeans	8	0	4.5
12	Vector Sums	1	0	1.2
13	Matmul	0	0	1.4
14	Pairwise sums	4	0	1.7
15	Cube	1	0	1.1
16	Square	1	0	1.1
17	Deadlock0	3	1	1.4
18	Deadlock2	0	1	1.3
19	Seive1	2	0	1.5

TABLE 5.2: Results: CUDA benchmarks verified using GPUVerify

possible data race. On the other hand, GKLEE reports only the first instance of a data race and terminates execution at that point.

An instance of a data race is found in the benchmarks **Cube**, **Square** and **Vector Sums**. This data race is context based i.e. it only occurs if there are multiple threads in a block. Else these kernels are free of bugs. Such bugs remain hidden when the configuration parameters are always set to the required values; in this case that value is 1 for both the number of threads and the number of blocks.

Another important observation can be made from benchmark 24, Loop4a. In this case, GPUVerify terminates with an unhandled exception. Due to lack of time and constraints imposed by the scope of our research, this error was not studied in detail.

5.2 Verification using GKLEE

Unlike GPUVerify, GKLEE is limited to CUDA applications. Consequently, only benchmarks written using CUDA which were complete with all the required libraries could be tested using GKLEE. Moreover, GKLEE is a sophisticated tool compared to GPUVerify. Its output is more detailed and takes time to interpret. But the results can be categorised into the following categories: Data Races, Deadlocks (Barrier Divergence), Rate of Memory Coalescing, Rate of warp divergence and Rate of bank conflicts. The following table lists the data from our tests.

Id 4.1	Benchmarks	Errors		Performance Bugs			Time
		Data Race #	BD # [2]	BCR % [3]	WDR % [4]	MCR % [5]	
10	Pairwise-sums timed	1	0	0	2, 45	100	1m 21.9s
11	GPU kmeans	1	0	0	0, 25	96, 75	1m 33.8s
12	Vector Sums	1	0	0	0	100	0m 0.9s
13	Matmul	1	0	0	50	100	0m 3.7s
14	Pairwise sums	1	0	0	50	100	0m 0.5s
15	Cube	1	0	0	0	100	0m 5.2s
16	Square	1	0	0	0	100	0m 3.4s
17	Deadlock0	0	1	NA	NA	NA	0m 1.4s
18	Deadlock2	0	1	0	50	100	0m 2.8s
19	Seive1	1	0	0	100	100	0m 6.3s
20	Simple-Error Handling	0	0	0	3, 100	100	4m 7.2s
21	Interblock race	1	0	0	0	100	0m 0.5s
22	Memory	0	0	0	20	100	0m 48.6s
23	Bank Conflict	0	0	100	0	100	0m 1.4s
25	SumMatrix-1D grid 2D block	Err	Err	Err	Err	Err	Timeout [6]
26	SumMatrix-2D grid 2D block	0	0	0	100	100	1m15.8s

Table 5.3: Results: Benchmarks verified using GKLEE

GKLEE reports the first occurance of a race condition and then terminates the program. This can be seen from the two tables listed above. While GPUVerify finds multiple data races in several benchmarks, GKLEE finds at most one. To verify this, a sample kernel

[2] Barrier Divergence, reported as a potential deadlock in GKLEE
[3] Bank Conflict Rate
[4] Warp Divergence Rate, with two sub-parts - Warp WDR and Barrier Interval (BI) WDR
[5] Memory Coalescing Rate, has two sub-divisions - Warp MCR and Barrier Interval (BI) MCR
[6] Timeout set at 80 mins; Benchmark 25 takes 84m17.8 seconds and is forcefully stopped

```
[GKLEE]: Under the pure canonical schedule, thread 0 and 1 incur a Write-Read
race (Actual) on
[GKLEE] Inst:
Instruction Line: 6, In File: dat_race.cpp, With Dir Path: /home/anmol/Thesis/
Example_Code
[File: /home/anmol/Thesis/Example_Code/dat_race.cpp, Line: 6, Inst:
      b[i] += b[i+1];]
  %16 = load i32* %15, align 4, !dbg !718
<R: 26003056, 4:2880154539, b0, t0>
[GKLEE] Inst:
Instruction Line: 6, In File: dat_race.cpp, With Dir Path: /home/anmol/Thesis/
Example_Code
[File: /home/anmol/Thesis/Example_Code/dat_race.cpp, Line: 6, Inst:
      b[i] += b[i+1];]
  store i32 %22, i32* %20, align 4, !dbg !718
<W: 26003056, 4:1465341782, b0, t1>
[GKLEE]: Under pure canonical schedule, a read-write race is found from BI 1 o
f the block 0
KLEE: ERROR: /home/anmol/Thesis/Example_Code/dat_race.cpp:20: execution halts
on encounering a (global) race
KLEE: NOTE: now ignoring this error at this location
```

FIGURE 5.1: GKLEE output for kernel A.5. Only the first data race is reported and execution terminates.

A.5 was written with two data races. On running GKLEE, the following results were found. Figure 5.1 illustrates an excerpt of the results.

Moreover GKLEE reports a potential deadlock in the benchmarks deadlock0 A.7 and deadlock2. The deadlock here is an occurrence of the case mentioned in 3.6.2. As can be seen, the **if** condition evaluates to **true** for all threads with **tid** less than 50 and **false** for threads 51 to 63. Consequently, the threads diverge into two different paths of execution, thus causing a deadlock.

Let us now consider benchmarks 25 and 26, both of which have the same kernel and compute the sum of two square matrices. The only difference is that benchmark 25 uses a 1 dimensional grid while benchmark 26 uses a 2 dimensional one when it calls the kernel. A major observation here is that GKLEE times out for benchmark 25. The reason being the size of the input matrix which is set to 16384, a significantly large value. The number of threads generated in this benchmark is directly proportional to the dimensions of the input matrix. Consequently, the kernel runs on a large number of threads, leading to a timeout (timeout value was fixed at 80 mins in these experiments). One can also observe that GKLEE executes normally for benchmark 26. The kernel is the same but it is called with different values for the configuration parameters. The size of the input matrix is set to 16 in this case. This relation between number of threads and time of execution is explored further in Graph 5.3 in section 5.3.

5.3 Comparative analysis of GPUVerify and GKLEE

In this section we compare the two tools on three parameters: Bugs reported by the tools, runtimes for benchmarks tested using both tools (common benchmarks) and variation in runtime on changing configuration parameters.

5.3.1 Difference in bugs reported

The two bugs that both GPUVerify and GKLEE report are data races and barrier divergence. However, due to the different methodology they follow, the tools provide differing results when tested on the same benchmarks. During the tests, data races were detected in nine of the ten common benchmarks. The following table 5.4 makes a comparative analysis of GPUVerify and GKLEE for the ten benchmarks that were tested using both the tools.

Id 4.1	Benchmark	Number of Data Races detected		Remarks
		GPU Verify	GKLEE	
10	Pairwise sums timed	4	1	GKLEE exits after first data race is detected
11	GPU Kmeans	8	1	GKLEE exits after first data race is detected
12	Vector sums	1	1	Data races occurs only if kernel is called with multiple threads
13	Matmul	0	1	GKLEE reports a benign data race
14	Pairwise sums	4	1	GKLEE exits after first data race is detected
15	Cube	1	1	Data race occurs only if kernel is called with multiple threads
16	Square	1	1	Data race occurs only if kernel is called with multiple threads
17	Deadlock0	3	0	GKLEE exits after reporting a potential deadlock (barrier divergence)
18	Deadlock2	0	0	Neither tool reports any data races
19	Seive1	2	1	GKLEE exits after first data race is detected

TABLE 5.4: Comparative analysis of data races reported by GPUVerify and GKLEE

Among the kernels that were tested, both tools report the same results for barrier divergence. GKLEE reports it as a potential deadlock whereas GPUVerify states that the barrier may be reached by non-uniform control flow.

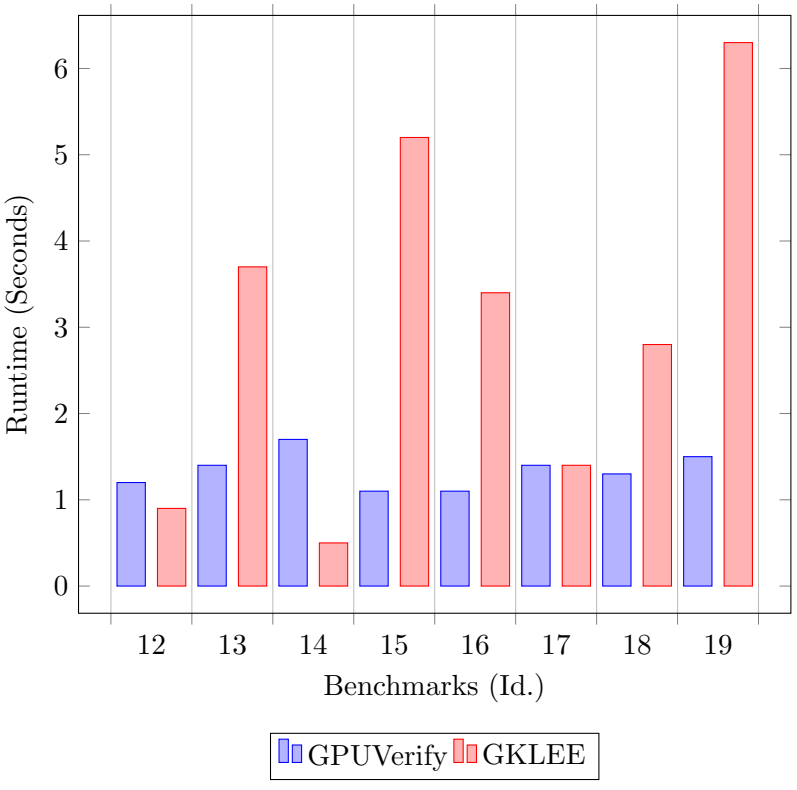
5.3.2 Analysis of execution time

The two chosen software, GPUVerify and GKLEE, were compared for three factors:

1. Runtimes for the benchmarks that were tested using both tools (common benchmarks)
2. Variation in runtime with respect to length of code for common benchmarks
3. Variation in runtime for a single benchmark (Pairwise sums) when number of threads per block are increased while keeping the number of blocks constant

Graph 5.1 compares the runtimes of GPUVerify and GKLEE for eight of the ten common benchmarks. As can be seen, GKLEE takes more time to complete its analysis than GPUVerify for five of the eight benchmarks. The remaining two benchmarks, pairwise sums timed and GPU kmeans were not included as the runtimes of GKLEE were too large to be represented in the same graph. For these benchmarks, the difference in runtime was even greater, as can be observed from 5.2 and 5.3.

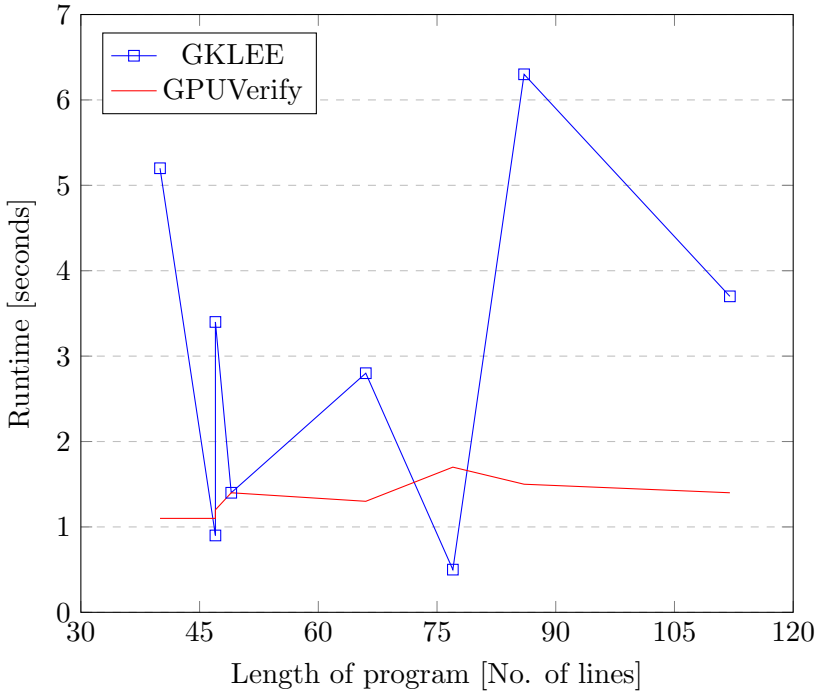
Graph 5.1: Comparison of runtimes for common benchmarks



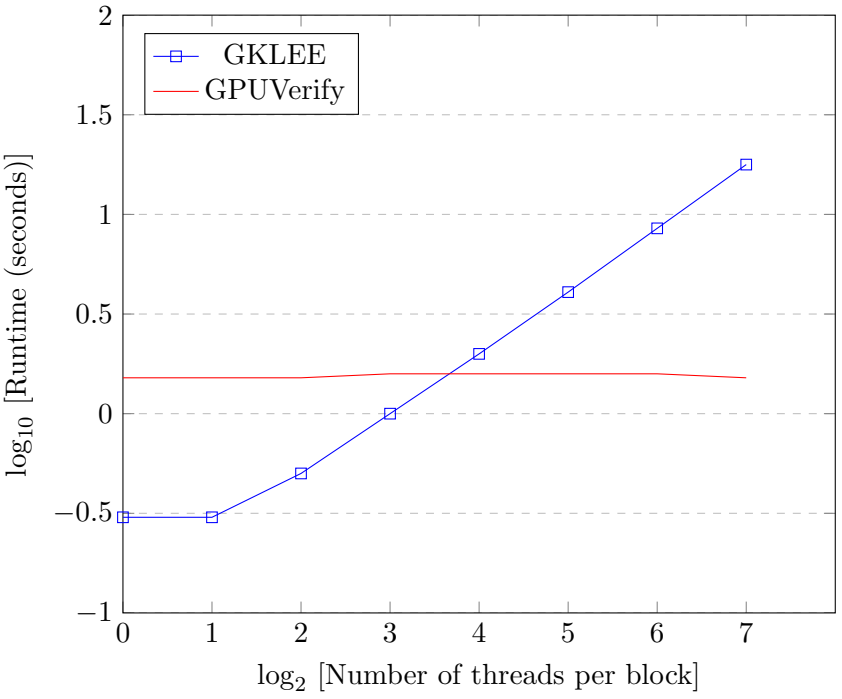
Graph 5.2 depicts the variation in runtime with respect to the length of the code. It must be noted that for such a small sample of relatively small CUDA kernels, we cannot make generalised conclusions from the trends we observe in this graph. However, what can be said is that the runtime of GKLEE varies significantly with respect to the length of the program. Contrastingly, the runtime of GPUVerify remains almost constant for the chosen benchmarks.

Graph 5.3 plots the variation in runtime of both tools when the configuration parameters, namely the number of blocks and number of threads per block are changed. The graph has a logarithmic scale with the x-axis showing the $log_2[Numberofthreadsperblock]$ and y-axis showing $log_10[Runtime(seconds)]$. In this experiment we hold the number of blocks in the grid to an constant value of two while varying the number of threads from 1 to 128 in the increasing powers of 2 i.e. the number of threads are 1, 2, 4, 8, 16, 32, 64 and 128 in the seven cases respectively. The experiment was conducted only on one benchmark, Pairwise sums.

Graph 5.2: Variation in runtime with length of code



Graph 5.3: Runtime of GKLEE and GPUVerify for Pairwise Sums



Chapter 6

Conclusion

The project has progressed according to the time line set at the beginning of the thesis, barring minor delays caused by delay in acquiring resources, installation of the verification tools, namely GPUVerify and GKLEE and resolving compatibility issues between kernels and the two verification tools. Over the course of this project, we have found errors that occur commonly in GPU kernels. We have selected twenty-six open-source benchmarks to represent both the OpenCL and CUDA paltforms. These have been tested using GPUVerify and GKLEE, the results were documented and analysed. The third verification tool, PUG, was not installed as it has been subsumed by GKLEE. This chapter outlines the conclusions that can be drawn from the work we have done during the course of this thesis.

Scope of the software The two software differ greatly in their scope. GPUVerify detects only two programming bugs, data races and barrier divergence while GKLEE also reports performance issues like bank conflicts, warp divergence and non-coalesced memory accesses. GPUVerify reports all potential data races and barrier divergence cases in a single barrier interval given kernel while GKLEE exits after reporting the first non-benign instance of any of these two bugs. Moreover, GPUVerify covers both the OpenCL and CUDA platforms while GKLEE only analyses CUDA programs written in C/C++. However, GKLEE subsumes GPUVerify in terms of the issues it detects and the volume of information it provides in the results. GKLEE can also be used to test the entire application when all required libraries and dependencies have been satisfied, thus allowing developers to evaluate real world performance. On the other hand, GPUVerify only conducts a static analysis of kernels in isolation and cannot comment on their performance when they are run in parallel.

Portability aspects GPUVerify is more portable across different platforms owing to its nightly build and docker versions which have few system requirements and require libraries and tools that can be installed relatively easily. In contrast, GKLEE is less portable, given that it needs tools that often take longer to install and can require a specific version of a given operating system. The installation procedure also leaves greater chance for error due to system inconsistencies or mistakes made by the user. The installation steps for both software are adequately documented on their respective web-pages and Github repositories.

Learnability and Usability issues GPUVerify is relatively much easier to run and the documentation and supporting literature on GPUVerify is adequate to resolve most issues faced during testing. This enables the user to use the tool efficiently and effectively. Comparatively, the GKLEE Github repository provides less documentation of the various modes in which GKLEE can be used. Moreover, the output of GPUVerify is simple and easy to interpret. However, in the case of GKLEE, the results generated are relatively complicated and detailed as it addresses many more aspects of the application than GPUVerify. This makes GPUVerify more usable and learn-able as a tool compared to GKLEE.

Error-free termination and execution time Both tools terminate for almost all the benchmarks, with a few notable exceptions. GPUVerify returns an error for benchmark Loop4a A.3. GKLEE exceeds the set time limit for benchmark 25, sumMatrix1Dgrid2Dblock. The reason for the excessive time taken are the large dimensions of the grid in that benchmark. In general, the execution time of GKLEE increases linearly with the values of the configuration parameters. This trend can be observed from Graph 5.3. Also, GKLEE takes longer to execute than GPUVerify for seven of the ten common benchmarks, as can bee seen from graph 5.1 and tables 5.2, 5.2 and 5.3. Moreover, its runtime depends significantly on the length of the code, although no direct relationship can be asserted between the two. GPUVerify, however, takes almost the same time for testing kernels that vary in length. This can be observed from graph 5.2.

Recommendations for usage Based on our analysis, GPUVerify is best suited to be used during the initial and intermediate phases of developing an application as it can analyse individual kernels without the need for a third-party libraries. While same can be done with GKLEE by writing a simple main function to call the kernel, GKLEE will report only the first instance of a bug. Also, the performance determined by GKLEE for an individual run of a kernel function can differ significantly with the performance of the same kernel when it is run along with the entire application. Consequently, GKLEE

can be used to test the application as a whole and gauge its real performance in the last phase of software development.

Finally, we can conclude that both GPUVerify and GKLEE provide much needed and useful mechanisms to detect programming bugs such as data races and barrier divergence. GKLEE also reports potential performance issues in the program by conducting a concolic analysis. There is potential for improvements with regards to usability and learn-ability aspects of both tools, especially for GKLEE.

6.1 Future work

This research project had a timeline of sixteen weeks. Th researcher had little or no prior knowledge of GPUs and GPU software. Consequently, the scope of this thesis was restricted to only the aspects mentioned in this report. This leaves open the opportunity to study these tools in greater detail.

One specific aspect is to carefully analyse the nature of computations in a specific benchmark and to group similar benchmarks together during the testing phase. Possible categories can include kernels that use floating point calculations and those that have nested loops. Another classification can be made based on libraries that an application is using to assess the effect of specific third party APIs on performance. Such a qualitative classification of benchmarks would enable researchers to study the performance of GPUVerify and GKLEE in greater depth.

Another important aspect that needs attention is the possibility of false positives and false negatives being reported by both tools. A false positive occurs when either of these tools reports a bug that does not exists in reality. In contrast, a false negative happens when the tool reports an actual case of a data race as benign or does not report it at all. Such issues can greatly reduce the reliability of a verification tool and therefore must be analysed in greater detail.

Lastly, we hope to present the results of this thesis to the developers of GPUVerify and GKLEE and seek their review of our analysis. Based on their feedback, we can further improve the qualitative aspects of our research, thus enabling a greater understanding of these verification software.

Appendix A

Kernel Source code

A.1 Loop4.cl

```
1  __kernel void foo(__global int *p, __global int *q) {
2    int i = get_global_id(0);
3
4      int size = get_global_size(0);
5
6      int f = 0;
7      for(int j = 0; j < size; j++) {
8      int N = p[i];
9      for(int k = 0; k < N; ++k) {
10       int M = q[i % j];
11       for(int l = 0; l < M; ++l) {
12         for(int m = 0; m < p[q[i]]; ++m) {
13           f += p[k] * p[j] / q[l] * sqrt((float)m);
14         }
15       }
16     }
17     }
18
19     p[i] = f;
20 }
```

Listing A.1: Kernel - Loop4.cl

A.2 N_Body_Computation.cu

```
#include <malloc.h>
#include <math.h>
#include <time.h>
#include <stdio.h>
struct Vector
{
    float x;
    float y;
    float z;
    __device__ float d_influenceBy(Vector p)
    {
      return 1;///sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y)+(z-p.z)*(z-p.z));
    }
    __host__ float h_influenceBy(Vector p)
    {
      return 1;///sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y)+(z-p.z)*(z-p.z));
    }
};

__host__ int ciel(float value)
{
  float mantissa = value - (int)value;
  return ((int)value + (mantissa==0 ? 0 : 1));
}

__global__ void forceComp(Vector *positions, int bodyCount, float*
    resultantForce, int bodiesPerThread)
{
  extern __shared__ float perBlockCache[];
  int tid = threadIdx.x*bodiesPerThread;
  int Limit = tid + bodiesPerThread;

  if( tid < bodyCount )
  {
    perBlockCache[threadIdx.x] = 0.0;
    while( tid < Limit )
    {
      if( blockIdx.x != tid )
        perBlockCache[threadIdx.x] += positions[blockIdx.x].d_influenceBy(
    positions[tid]);
      tid++;
    }
    __syncthreads();

    /* now do reduction by addition for the resultant
     * force on body with Id = blockIdx.x */
```

```
45      int reduceDim = blockDim.x/2;
46      while(reduceDim>0)
47      {
48        if( threadIdx.x < reduceDim )
49      perBlockCache[threadIdx.x] += perBlockCache[threadIdx.x+reduceDim];
50        __syncthreads();
51        reduceDim /= 2;
52      }
53      if(threadIdx.x == 0)
54        resultantForce[blockIdx.x] = perBlockCache[0];
55    }
56  }
```

LISTING A.2: Kernel - N_Body_Computation.cu

A.3 Loop4a

```
1  __kernel void foo(__global int *p, __global int *q) {
2    int i = get_global_id(0);
3
4      int size = get_global_size(0);
5
6      int f = 0;
7      for(int j = 0; j < size; j++) {
8      int N = p[i];
9      if(N >= size) return;
10     for(int k = 0; k < N; ++k) {
11       int M = q[i % j];
12       if(M >= size) return;
13       for(int l = 0; l < M; ++l) {
14         if(p[q[i]] >= size) return;
15         /*for(int m =0; m<10;m++){
16                 f=2;
17         }*/
18         for(int m = 0; m < p[q[i]]; ++m) {
19           f += p[k] * p[j] / q[l] * 1;//sqrt((float)m);
20           if(f >= size * size) return;
21         }
22       }
23     }
24     }
25
26     p[i] = f;
27 }
```

LISTING A.3: Kernel - Loop4a.cl GPUVerify terminates with error for this benchmark

A.4 Inter_Block_Race.cu

```
1 #define N 128
2 #define B 2
3
4 __global__ void k(int* in)
5 {
6   in[threadIdx.x] = blockIdx.x;
7 }
8
9 int main()
10 {
11   int* in = (int*) malloc(N * sizeof(int));
12   int* din;
13   cudaMalloc((void**) &din, N*sizeof(int));
14   k<<<B,N/B>>>(din);
15 }
```

Listing A.4: Kernel - Inter_block_data.cu race

A.5 Data Race example in CUDA

```
1 #define N 16
2 __global__ void add(int* a, int* b){
3         int i = threadIdx.x + blockIdx.x * blockDim.x;
4         if(i<N-1){
5                 b[i] += b[i+1];
6                 a[i] += a[i+1];
7         }
8 }
9
10 int main(){
11         int* a = (int*) malloc(N * sizeof(int));
12         int* d_a;
13         cudaMalloc((void**) &d_a, N*sizeof(int));
14         int* b = (int*) malloc(N * sizeof(int));
15         int* d_b;
16         cudaMalloc((void**) &d_b, N*sizeof(int));
17         add<<<4, 4>>>(d_a,d_b);
18 }
```

Listing A.5: Kernel - CUDA example with two data races

A.6 Cube.cu

```
#include <stdio.h>
__global__ void cube(float * d_out, float * d_in) {
  int idx = threadIdx.x;
  float f = d_in[idx];
  d_out[idx] = f * f * f;
}
int main(int argc, char ** argv) {
  const int ARRAY_SIZE = 96;
  const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

  //generate the input array on the host
  float h_in[ARRAY_SIZE];
  for (int i = 0; i < ARRAY_SIZE; i++) {
    h_in[i] = float(i);
  }
  float h_out[ARRAY_SIZE];

  //declare GPU memory pointers
  float * d_in;
  float * d_out;
  //allocate GPU memory
  cudaMalloc((void**) &d_in, ARRAY_BYTES);
  cudaMalloc((void**) &d_out, ARRAY_BYTES);
  //transfer the array to the GPU
  cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);

  //launch the kernel
  cube<<<1, ARRAY_SIZE>>>(d_out, d_in);

  //copy back the result array to the CPU
  cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);
  //print out the resulting array
  for (int i = 0; i < ARRAY_SIZE; i++) {
    printf("%f", h_out[i]);
    printf(((i % 4) != 3) ? "\t" : "\n");
  }
  cudaFree(d_in);
  cudaFree(d_out);
  return 0;
}
```

Listing A.6: Kernel - Cube.cu

A.7 Deadlock0.cu

```
#include <cstdio>
#define N 50
#define B 2
#define T 32

__global__ void dl(int* in)
{
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if(tid < N)
  {
    if(in[tid] % 2 == 0)
      in[tid]++;

    __syncthreads(); // ouch

    int sum = in[tid];
    if(tid > 0){
      sum += in[tid-1];

    }if(tid < N - 1)
      sum += in[tid+1];
    in[tid] = sum / 3;
  }
}

int main()
{
  int* in = (int*) malloc(N*sizeof(int));
  for(int i = 0; i < N; i++)
    in[i] = i;

  int* din;
  cudaMalloc((void**)&din, N*sizeof(int));
  cudaMemcpy(din, in, N*sizeof(int), cudaMemcpyHostToDevice);

  dl<<<B,T>>>(din);

  cudaMemcpy(in, din, N*sizeof(int), cudaMemcpyDeviceToHost);

  for(int i = 0; i < N; i++)
    printf("%d ", in[i]);
  printf("\n");
  free(in); cudaFree(din);
}
```

Listing A.7: Kernel - Deadlock0.cu

Bibliography

[1] CUDA v7.5 Toolkit Documentation. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz46u9vzDb1.

[2] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: a verifier for gpu kernels. In *ACM SIGPLAN Notices*, volume 47, pages 113–132. ACM, 2012.

[3] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P Rajan. Gklee: Concolic verification and test generation for gpus. In *ACM SIGPLAN Notices*, volume 47, pages 215–224. ACM, 2012.

[4] Ethel Bardsley and Alastair F. Donaldson. *NASA Formal Methods: 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 – May 1, 2014. Proceedings*, chapter Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels, pages 230–245. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06200-6. doi: 10.1007/978-3-319-06200-6_18. URL http://dx.doi.org/10.1007/978-3-319-06200-6_18.

[5] Wei-Fan Chiang, Ganesh Gopalakrishnan, Guodong Li, and Zvonimir Rakamarić. *NASA Formal Methods: 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, chapter Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-Bounding, pages 213–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-38088-4. doi: 10.1007/978-3-642-38088-4_15. URL http://dx.doi.org/10.1007/978-3-642-38088-4_15.

[6] Guodong Li and Ganesh Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196. ACM, 2010.

[7] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[8] Khronos OpenCL Working Group et al. The opencl specification. *version*, 1(29):8, 2008.

[9] CUDA Nvidia. Compute unified device architecture programming guide. 2007.

[10] Kate Gregory and Ade Miller. C++ amp: accelerated massive parallelism with microsoft visual c++. 2014.

[11] CUDA Nvidia. Programming guide, 2008.

[12] Heterogeneous programming with gpus. URL http://apos-project.eu/Tools-Technologies/heterogeneous-programming-with-gpus.html.

[13] Tom Goddard. Gpu computing. URL https://www.cgl.ucsf.edu/chimera/data/group-meeting-dec2008/gpu.html.

[14] John Owens et al. Gpu computing. URL http://cs.utsa.edu/~qitian/seminar/Spring11/03_04_11/GPU.pdf.

[15] Wen-Mei Hwu. Three Challenges in Parallel Programming. 2012. URL http://parallel.illinois.edu/blog/three-challenges-parallel-programming.

[16] Igor Ostrovsky. How gpu computing cam to be used for general purpose, igor ovstrosky blog. URL http://igoro.com/archive/how-gpu-came-to-be-used-for-general-computation/.

[17] Randall Head. What's the big deal with gpgpus? URL http://www.vizworld.com/2009/05/whats-the-big-deal-with-cuda-and-gpgpu-anyway/#sthash.bGL6wt6A.dpbs.

[18] Brian O'Sullivan. A quick programmer's look at nvidia's cuda. 2007. URL http://www.serpentine.com/blog/2007/02/22/a-quick-programmers-look-at-nvidias-cuda/.

[19] What is so hard about non-graphic programming on a gpu? URL http://arstechnica.com/uncategorized/2007/02/8931/.

[20] Lawrence Latif. AMD thinks most programmers will not use CUDA or OpenCL, The Inquirer, 2013. URL http://www.theinquirer.net/inquirer/news/2257035/amd-thinks-most-programmers-will-not-use-cuda-or-opencl.

[21] Forked repository from original github repositories of Leiming Yu. URL https://github.com/Anmol-007/oclKernels.

[22] Github repository of eric holk. URL https://github.com/eholk/opencl-stress.

[23] Github repository of pradeep garigipati. URL https://github.com/9prady9/CUDA.

[24] GitHub repository of Chen Rudan. URL https://github.com/chenrudan/cuda_examples.

[25] Github repository of francesco caruso. URL https://github.com/fcaruso/CudaImageBlur.

[26] Github repository of will landau (materials for the iowa state university statistics department fall 2012 lecture series on general purpose gpu computing). URL https://github.com/wlandau/gpu.

[27] Github repository of chiranth siddappa. URL https://github.com/chiranthsiddappa/GPU.

[28] Official test cases provided by GKLEE developers, . URL https://github.com/Geof23/GkleeTests.

[29] Github repository of Tomasz Gasior. URL https://github.com/Tom-Demijohn/CUDA_programs.

[30] Github repository of Carl. URL https://github.com/daxiongshu/cuda_textbook_examples.

[31] Installation Procedure, GPUVerify documentation page, Multicore programming Group, Imperial College London. URL http://multicore.doc.ic.ac.uk/tools/GPUVerify/docs/installation.html.

[32] Official page of the Python programming platform. URL https://www.python.org/.

[33] Official page of pip Python package installer. URL https://pip.pypa.io/en/stable/.

[34] GitHub page of psutil Python library. URL https://github.com/giampaolo/psutil.

[35] Official page of Mono project. URL http://www.mono-project.com/.

[36] Official page of GNU Compiler Collection (GCC). URL https://gcc.gnu.org/.

[37] Vern Paxson et al. Flex–fast lexical analyzer generator. *Lawrence Berkeley Laboratory*, 1995.

[38] Charles Donnelly and Richard Stallman. *Bison: The YACC-compatible parser generator*. Free Software Foundation Cambridge (MA) 02139, 1992.

[39] Ken Martin and Bill Hoffman. *Mastering CMake*. Kitware, 2015.

[40] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.

[41] Official page of Prover of User GPUs. URL http://formalverification.cs.utah.edu/PUG/.

[42] Homepage of the developers of GKLEE, the Gauss Research Group, Department of Computer Science, university of Utah, . URL http://formalverification.cs.utah.edu/GKLEE/.

[43] Official GitHub repository of GKLEE , . URL https://github.com/Geof23/Gklee.