2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)

A Comparative Study of GPUVerify and GKLEE

Anmol Panda
Department of Computer Science
and Information Systems
BITS Pilani K. K. Birla Goa Campus
Goa, India
Email: f2012123@goa.bits-pilani.ac.in

Philipp Rümmer
Department of Information Technology
Uppsala University
Sweden
Email: philipp.ruemmer@it.uu.se

Neena Goveas
Department of Computer Science
and Information Systems
BITS Pilani K. K. Birla Goa Campus
Goa, India
Email: neena@goa.bits-pilani.ac.in

***Abstract*—Use of Graphics Processing Unit (GPU) software is increasing due to the need for data intensive operations and availability of GPUs. This has led to a need for effective GPU software verification tools. These tools have to satisfy requirements such as accuracy, reliability and ease of use. In this work, we have considered two such tools: GPUVerify and GKLEE. Our objectives were to learn about the common challenges developers faced in GPU programming, to understand the specific bugs that these two tools report and compare their scope and scalability aspects. We have also considered usability and learnability aspects. 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. The conclusions have been included in the final section.**

***Keywords*—GPU kernel verification, GPUVerify, GKLEE**

I. Introduction

Given the rapid developments in multi-core processor technology, GPUs now play a vital role in various types of applications that rely on these chips for parallel computation [1], [2], [3], [4], [5], [6], [7]. 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, incorrectly placed barriers and inefficient memory accesses. Such errors, if unchecked, can render the system in an undefined and unpredictable state [8], [9], [10]. 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 [9] and GKLEE [10] have been developed. These tools vary in their target applications, the scope of errors they report and their approach and depth of testing. These tools have to identify various bugs such as data races and barrier divergence. In addition dynamic aspects such as thread divergence within a warp, inefficient memory accesses and bank conflicts are reported by GKLEE.

In this work, we have studied some of the common errors that can occur in GPU software. We analysed the tools, GPUVerify and GKLEE, by studying their performance on 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.

A. Scope and Objectives

The objectives of the research were to analyse these software for their similarities and differences as well as benefits and disadvantages of using each. The scope of the work was limited to the usage and benefits of deploying the said tools with a specific focus on 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 sections, we have described the various stages of this work. Firstly, section two outlines the broad challenges in GPU computing paradigm. Section three documents the experimental setup and process while section four lists the results and analysis of those experiments. Finally, section five explains the conclusions of our research and the potential for future work in this area.

B. Related works

The multi-domain utility of massively parallel GPUs has motivated researchers to address the need for verifying GPU software written for myriad sectors. Betts et al [9] describe GPUverify, the background behind its need and development, the mathematical model used for verification and the performance of the tool. Ethel Bardsley and Alastair Donaldson [11] 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 [10] describe the logical model of GKLEE, its capacity to detect bugs and performance issues and its performance during verification of select commercial SDKs. T Wei-Fan Chiang et al [12] describe a new method to detect bugs utilising both barriers and atomics.

II. Challenges in GPU Computing

Several major challenges emerge in designing and programming software that can fully harness the computing power of these GPUs.

Firstly, the GPU programming model is parallel in nature. Inefficient use of parallel APIs by novice programmers [13] [14] can degrade performance of General Purpose GPU

978-1-5090-3669-1/16/$31.00 ©2016 IEEE

(GPGPU) applications and lack of backward and forward API compatibility can make them non-viable in the long term [15]. Also, not all problems have massively parallel algorithms [16]. Secondly, in the absence of a dedicated cache memory for each processor [13], non-coalesced memory accesses [17], data transfers between multiple layers of memory and deploying too few or too many threads adversely impact performance. Moreover, GPU computing is yet to achieve application portability with other parallel domains such as multi-core CPUs and stream processors [15]. Thirdly, optimizing performance for GPUs is a difficult process in the absence of sophisticated control logic for branch and loop prediction. Since data transfer remains the slowest part of the process [18], GPGPU applications must be profiled to locate performance bottlenecks and quantify their overhead. Using too many third party libraries to optimize code can make the application less modular, less portable and difficult to debug [13]. In addition, minor changes in the program can cause performance cliffs [15]. Fourthly, achieving stated performance requires significant time and effort. In the CUDA framework, many applications may only achieve about half the expected performance expected from the CUDA application and few would exceed 10 percent of their peak performance as data transfers across memory levels are not automated in the compiler [19]. Correcting such flaws requires significant investment in time and resources on a scale that only well-funded research organisations can deliver [19] [20] [21]. Lastly, not all applications require the degree of speedup that a GPU can deliver while others may need much time and effort to develop and market, thus reducing the pool of potential developers. These factors adversely impact the popularity of GPU programming as choice for research, especially among students.

A. 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 this work, we have considered certain programming and some performance bugs that can occur in GPU software. These include data races and incorrectly placed barriers.

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 and at least one of them is writing to it. In such a scenario, the outcome of the memory accesses is undefined. Consequently, the state of the application and by extension, that of the system is unpredictable.

Listing 1. Data Race example

```
__global__
void addToNextKernel(int *a,int b, int n){
    int i = threadIdx.x +
            blockDim.x*blockIdx.x;
    if( i < n )
        a[i+1] = a[i] + b;
}
```

Listing 1 is a common example of a data race [22]. Consider threads τ_1 and τ_2 with $i = 1$ and $i = 2$ respectively. When τ_1 executes, 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 and the result is unpredictable. In the example above, all threads τ_1 to τ_{n-1} have data races. In general, some data races may be actual, others could be benign while some occur only for certain values of configuration parameters.

2) Barrier Divergence: The next issue that we consider is barrier divergence. Barriers are introduced to synchronise threads to prevent bugs such as data races. The **__syncthreads()** and **barrier()** function calls are used in CUDA and OpenCL respectively to synchronise threads. 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**.

Listing 2. OpenCL example of Barrier Divergence and Data Race

```
__kernel void addEvenSubtractOdd
(__global int *numbers,
int n, __global int *sum){

    int res=0;
    int i, temp;
    i = get_local_id(0);
    if(i<n){
        temp = numbers[i];
        if(temp%2 == 0){
            res += temp;
            barrier(CLK_GLOBAL_MEM_FENCE);
        }else{
            res -= temp;
            barrier(CLK_GLOBAL_MEM_FENCE);
        }
    }
    *sum = res;
}
```

A common instance of barrier divergence occurs when barriers are within the scope of conditional statements. In such a case, if some threads skip the scope of the condition while others wait at the barrier inside the scope, the waiting threads will never be released, thus leaving the system in an unstable and unpredictable state. An instance of such a case is shown in Listing 2. The threads that operate on even data items wait at the first barrier while the odd ones wait at the second barrier. In such a scenario, the output is undefined with the possibility of unintended side-effects [1]. Therefore, although 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.

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, but also locates any bank conflicts, warp divergences and non-coalesced memory accesses through a concolic analysis of the kernel.

[1] Section 12.4, "Synchronising Divergent Threads in a Group", CUDA Toolit Documentation [8]

III. Experiments

A. Benchmarks

We chose twenty-six benchmarks [23], [24], [25], [26], [27], [28], [29], [30], [31] from open source applications available on GitHub. These applications cover a wide variety of domains such as image processing, data mining tools, mathematical operations, etc. Some benchmarks were chosen from the list of examples provided by the developers of GKLEE [10]. This enabled us to understand the behaviour of GKLEE in greater detail.

B. Experimental Setup

The benchmarks were then tested for bugs using the chosen verification tools, GPUVerify and GKLEE. The experiments were conducted on a system detailed in Table I.

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 I. System Specifications

C. Results

In this section, we have discussed the results of our experiments. We have tabulated the number of potential bugs reported and the execution time. Tables II and III show the results for GPUVerify with OpenCL and CUDA benchmarks respectively. Table IV lists the results of tests conducted with GKLEE. Table V includes a comparative analysis of GPUVerify and GKLEE.

Id	Benchmark	# DR [2]	# BD [3]	Time (secs)
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 [4]	Err	17.8s

TABLE II. Results: OpenCL benchmarks verified using GPUVerify

In each of the cases, except N-Body Computation, GPUVerify completes its execution in less than ten seconds. Moreover, GPUVerify reports each possible data race within a barrier interval. On the other hand, GKLEE reports only the first instance of a data race and terminates execution at that point.

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, the output of GKLEE is more detailed and takes time to interpret. Its 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	Benchmark	# DR	# BD	Time (secs)
6	N-Body Computation	2	2	39.7s
7	PI Estimation	3	0	3.9s
8	MatrixMultiply2	8	0	6.7s
9	Image Blur	0	0	0.7s
10	Pairwise sums timed	4	0	1.6s
11	GPU kmeans	8	0	4.5s
12	Vector Sums	1	0	1.2s
13	Matmul	0	0	1.4s
14	Pairwise sums	4	0	1.7s
15	Cube	1	0	1.1s
16	Square	1	0	1.1s
17	Deadlock0	3	1	1.4s
18	Deadlock2	0	1	1.3s
19	Seive1	2	0	1.5s

TABLE III. Results: CUDA benchmarks verified using GPUVerify

GKLEE reports the first occurrence of a race condition and then terminates the program. This can be seen from tables IV and V . While GPUVerify found multiple data races in several benchmarks, GKLEE found at most one.

We compared 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 IV.

IV. 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.

[2]Data Race

[3]Barrier Divergence

[4]GPUVerify exits with error: unhandled exception

[5]Barrier Divergence, reported as a potential deadlock in GKLEE

[6]Bank Conflict Rate

[7]Warp Divergence Rate, with two sub-parts - Warp WDR and Barrier Interval (BI) WDR

[8]Memory Coalescing Rate, has two sub-divisions - Warp MCR and Barrier Interval (BI) MCR

[9]Timeout set at 80 mins; Benchmark 25 takes 84m17.8 seconds and is forcefully stopped

Id	Benchmarks	Errors		Performance Bugs			Time
		DR #	BD # [5]	BCR % [6]	WDR % [7]	MCR % [8]	
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 [9]
26	SumMatrix-2D grid 2D block	0	0	0	100	100	1m15.8s

TABLE IV. RESULTS: BENCHMARKS VERIFIED USING GKLEE

A. 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. A comparative analysis of GPUVerify and GKLEE for the ten benchmarks that were tested using both the tools is included in Table V.

Id	Benchmark	Data Races #		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 V. 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.

B. 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

Figure 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. For one benchmark, the runtimes of both tools are equal. 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 III and IV.

Figure 5.2 depicts the variation in runtime with respect to the length of the code for seven of the ten common benchmarks. 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.

Figure 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 $log_2[ThreadsPerBlock]$ 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, namely Pairwise sums. It can be observed that the runtime of GPUVerify remains constant while the runtime of GKLEE increases linearly on the logarithmic scale.

Figure 5.1: Comparison of runtimes for common benchmarks

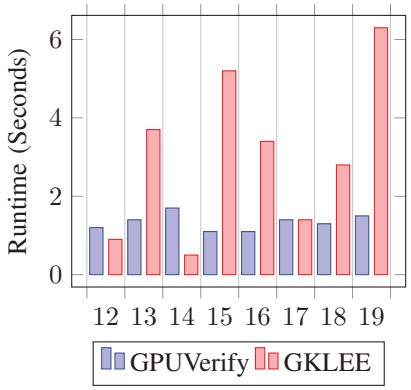


Figure 5.2: Variation in runtime with length of code

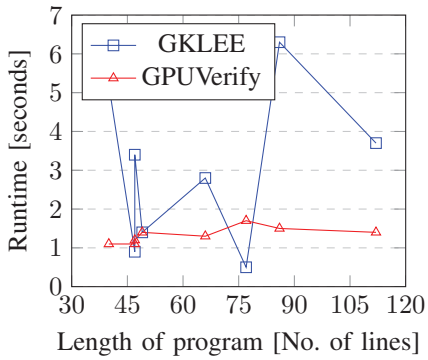
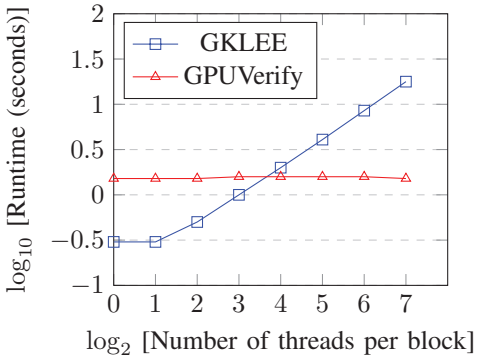


Figure 5.3: Comparison of runtime for Pairwise Sums



V. Conclusion

The following are the major conclusions of our research.

Scope of the software: GPUVerify detects all potential data races and diverging barriers in one barrier interval and it covers OpenCl and CUDA applications. On the other hand, GKLEE also reports bank conflicts, warp divergence and memory coalescing rates. It reports the first instance of data race or barrier divergence and then terminates. GPUVerify analyses kernels in isolation while GKLEE can test an entire application.

Portability, Learnability and Usability issues: GPUVerify is more portable than GKLEE since it requires libraries that are easy to install, has fewer system dependencies and almost no dependency on the version of OS used. It is easier to run and the documentation and supporting literature on GPUVerify is adequate to resolve most issues faced during testing. Comparatively, the GKLEE Github repository provides less documentation. Moreover, the output of GPUVerify is simple and easy to interpret while the results generated by GKLEE are relatively complicated and detailed as it addresses many more aspects of the application than GPUVerify.

Error-free termination and execution time: Both tools terminate for almost all the benchmarks, with a few notable exceptions. In general, the execution time of GKLEE increases linearly with the values of the configuration parameters as observed from Graph 5.3. Also, GKLEE takes longer to execute than GPUVerify for seven of the ten common benchmarks, as can be seen from graph 5.1 and tables III and IV. Unlike GPUVerify, the runtime of GKLEE varies with the length of the code, although no co-relation can be asserted. GPUVerify 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 while GKLEE can be used to test the entire application and gauge its real performance in the final phases of software development.

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.

VI. Future work

In the future, we would be categorising benchmarks by the nature of computations such as floating point calculations and nested loops or third party libraries used by the kernels. In this way, we can study the performance of these tools with respect to qualitative aspects of applications. Secondly, we will also look at the possibility of false positives and negatives reported. These issues can reduce the reliability of these tools and must be analysed in greater depth.

Acknowledgment

The first author would like to thank the Erasmus Mundus NAMASTE programme for funding the exchange studies at Uppsala University. During this period, the study was conceptualised and planned.

References

[1] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[2] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[3] W. H. Wen-Mei, *GPU Computing Gems Emerald Edition*. Elsevier, 2011.

[4] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *Micro, IEEE*, vol. 28, no. 4, pp. 13–27, 2008.

[5] J. Nickolls and W. J. Dally, "The gpu computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, 2010.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[7] E. Alerstam, T. Svensson, and S. Andersson-Engels, "Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration," *Journal of biomedical optics*, vol. 13, no. 6, pp. 060 504–060 504, 2008.

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

[9] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: a verifier for gpu kernels," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 113–132.

[10] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "Gklee: Concolic verification and test generation for gpus," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 215–224.

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

[12] W.-F. Chiang, G. Gopalakrishnan, G. Li, and Z. Rakamarić, *NASA Formal Methods: 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-Bounding, pp. 213–228. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38088-4_15

[13] Iakovos Panourgias, (2012), Heterogeneous programming with GPUs. [Online]. Available: http://apos-project.eu/Tools-Technologies/heterogeneous-programming-with-gpus.html

[14] T. Goddard. Gpu computing. [Online]. Available: https://www.cgl.ucsf.edu/chimera/data/group-meeting-dec2008/gpu.html

[15] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[16] Wen-Mei Hwu, "Three Challenges in Parallel Programming," 2012. [Online]. Available: http://parallel.illinois.edu/blog/three-challenges-parallel-programming

[17] I. Ostrovsky. (2010) How GPU Computing cam to be used for general purpose, Igor Ovstrosky Blog. [Online]. Available: http://igoro.com/archive/how-gpu-came-to-be-used-for-general-computation/

[18] R. Head. (2009) What's the big deal with gpgpus? [Online]. Available: http://www.vizworld.com/2009/05/whats-the-big-deal-with-cuda-and-gpgpu-anyway/#sthash.bGL6wt6A.dpbs

[19] B. O'Sullivan, "A quick programmer's look at nvidia's cuda," 2007. [Online]. Available: http://www.serpentine.com/blog/2007/02/22/a-quick-programmers-look-at-nvidias-cuda/

[20] John Stokes, (2007), What is so hard about non-graphic programming on a GPU? [Online]. Available: http://arstechnica.com/uncategorized/2007/02/8931/

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

[22] G. Li and G. Gopalakrishnan, "Scalable smt-based verification of gpu kernel functions," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 187–196.

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

[24] Github repository of eric holk. [Online]. Available: https://github.com/eholk/opencl-stress

[25] Github repository of pradeep garigipati. [Online]. Available: https://github.com/9prady9/CUDA

[26] GitHub repository of Chen Rudan. [Online]. Available: https://github.com/chenrudan/cuda_examples

[27] Github repository of francesco caruso. [Online]. Available: https://github.com/fcaruso/CudaImageBlur

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

[29] Github repository of chiranth siddappa. [Online]. Available: https://github.com/chiranthsiddappa/GPU

[30] Github repository of Tomasz Gasior. [Online]. Available: https://github.com/Tom-Demijohn/CUDA_programs

[31] Github repository of Carl. [Online]. Available: https://github.com/daxiongshu/cuda$_t extbook_e xamples$