Building a Dynamic Analyzer for CUDA based System.

SALAH T. ALSHAMMARI
salahtshammari@gmail.com
King Abdul-Aziz University, College of Computing and Information Technology, Jeddah, Saudi Arabia

Summary
The utilization of GPUs on general-purpose computers is currently on the rise due to the increase in its programmability and performance requirements. The utility of tools like NVIDIA's CUDA have been designed to allow programmers to code algorithms by using C-like language for the execution process on the graphics processing units GPU. Unfortunately, many of the performance and correctness bugs will happen on parallel programs. The CUDA tool support for the parallel programs has not yet been actualized. The use of a dynamic analyzer to find performance and correctness bugs in CUDA programs facilitates the execution of sophisticated processes, especially in modern computing requirements. Any race conditions bug it will impact of program correctness and the share memory bank conflicts to improve the overall performance. The technique instruments the programs in a way that promotes accessibility of the memory locations accessed by different threads well as to check for any bugs in the code of a program. The instrumented source code will be used initiated directly in the device emulation code of CUDA to send report for the user about all errors. The current degree of automation helps programmers solve subtle bugs in highly complex programs or programs that cannot be analyzed manually.

Keywords:
CUDA programs, GPU, Dynamic Analyzer, software testing, Race conditions, bank conflicts.

1. Introduction

After reaching the epitome of optimizing single-core processors, the microprocessor industry has shifted into integrating multiple cores to attain significantly high processing speeds. The application has integrated microprocessor cores has overwhelmingly transformed the computing industry. For instance, the General Processing Units (GPUs) are some of the computing units that use up to 128 cores, which leads to tremendous processing speed. This level of integration is regarded to as "manycore", a modern technology that has improved the computing platform to suit the high demand in the industry (Li, Guodong, and Ganesh [6]). Computer developers and programmers have been able to attain high concurrency in the execution of basic and sophisticated computer applications. The main goal of design GPUs is the effective executive of 3D rendering applications; however, there is a significantly high demand for advanced programmability by graphics designers and programmers. Consequently, they are utilized in the general multipurpose structures especially where complex instruction sets and rich memory hierarchies are needed. Crucially, it is through the use of CUDA that programmers can develop general purpose applications for multipurpose utilization of computer resources. Nonetheless, the risk of data races and the possibility of contention in the utilization of resources make it difficult to write multi-threaded GPU programs. Apparently, with just two threads the bug of synchronization in the CPU program may have a significantly low likelihood of occurrence when the sophisticated program executed.

A bug parallel program may have too many threads with a much higher possibility of happening. The use of debugging and testing tools used mainly for applications of the desktop are normally not available in a parallel environment with several processes. Therefore, in the parallel program, solving and locating a bug becomes very difficult. In a sequentially executing program, however, it is possible to locate and solve bugs using a small number of threads from a multi-threaded application. Apparently, locating and solving these bugs becomes impossible when hundreds or thousands are involved. At this point, the use of automated analyses enhances the handling of bugs on parallel programs.

The adaptation of software instrumentation techniques to CUDA programs. Certainly, the adaptation is done using modern and advanced instrumentation techniques since CUDA programs use multi-threaded barriers that have memory and space limitations. Traditional approaches are also written in simple programming languages, which make them incompatible with advanced programs. The use of an automated approach to instrumenting CUDA programs is highly efficient in detecting most classes of bugs during program runtime. The approach is recommended to programmers as a usable tool in finding performance bugs and correctness in complex codes which are too difficult to be analyzed manually. The automated approach has been successfully configured to automatically detect ineffective memory access patterns and race conditions bugs in CUDA programs. Mostly, race conditions and ineffective memory access patterns are representative of general classifications of codes problems, memory contentions that happen as thread counts, and core counts grow during the execution process and synchronization errors.

Manuscript received August 5, 2023
Manuscript revised August 20, 2023
https://doi.org/10.22937/IJCSNS.2023.23.8.9

2. The GPU

The performances of Graphics Processing Units continue to increase with the advancement of parallel microprocessor technology. Its peak performance speed has already bypassed the speed of Intel's most advanced CPU. Therefore, CPU's are most preferable in providing the most needed rapid performance growth in most high-demand computing requirements. The GPU performance technology utilizes replicating simple processing elements (PEs), which target throughput rather than single-thread performance. The system is also designed to allocate much less die area to caches and control logic. Additionally, different groups of PEs are harnessed under the SIMD control, which enables the system to amortize the area overhead of the instruction store and control logic.

Advanced cache management has been successfully utilized to increase the performance of the GPU by reducing memory latency. In this regard, the GPUs depend on massive multi-threading where the system supports up to thousands of threads. Apparently, constant tests have indicated that it is nearly impossible to reduce memory latency by merely increase the cache memory for bug processing while other processes are ongoing. Multi-threading technology increases performance by reducing the number of on-hold processes, which enhances increased multi-level performance.

Li et al. proposed that the automated abstraction of techniques can be significant in reducing the efforts required to undertake concurrent system analysis [6]. A high-precision approach to behavioral symmetry present in GPU programs facilitates CUDA race detection in a highly simplified approach. The use of abstraction techniques enhances a controlled flow of threads such that the debugging process takes less time as possible. An extensive analysis of complexity has played a significant role in addressing the issues required to articulate the complexity of locating and debugging errors during the execution of programs.

Programmers and advanced computer users can develop GPU graphics easily using various classical image processing algorithms using the Computer Unified Device Architecture (Yang, Yating, and Yong [12]). The main features of CUDA-GPU enhance the summary of the general process of enhancing the effectiveness of software development. CUDA enables programmers to implement several image processing algorithms such as histogram equalization, edge detection, and removal of clouds. An increase in the size of the image prompts a histogram computation leading to a speed-up of 40 times. Other image processing functions such as removal of clouds may prompt speed of about 70 times while edge detection may prompt a speed increment of a speed of about 400 times. These operations have made a significant transformation in the image processing requirements in a multidimensional computation framework.

1.1 General-Purpose Computation on GPUs

The computation performance of general-purpose on graphics processing unit GPUs has gained extensive interest following the flexibility and performance of GPUs with respect to the contemporary CPUs. GPGPU programs were previously written using graphics APIs. The GPU-specific hardware offered a reliable and powerful computing platform, which facilitated the execution of overhead non-graphics computation into graphics API. With time, significant advancements have been accomplished. Already, NVIDIA and ATI have released a series of software tools for simplifying the development of GPGPU applications. The Close-to-the-Metal (CTM), released by API in 2006 offers a low-level interface for programming of graphics processing unit GPU (Sanders and Edward [7]). Alternately, NVIDIA has released Tesla, Quadro, and GeForce based products, which are embedded in the architecture of Tesla hardware that allows programmers to implement general-purpose programs for the graphics processing units.

3. The CUDA

The CUDA NVIDIA is a development toolkit and a free source language. It is highly preferred due to it makes the processing of programming general-purpose applications simple for the latest GPUs NVIDIA.

CUDA is an application programming interface comprising an extended version of C with a runtime library. The characteristic abstractions provide a notion of a kernel function with a series of routines, which are invoked whenever the system is attempting to invoke commands during thread instances [16]. Thread instances comprise SIMD core and barrier synchronization. These features are highly dependent on a series of debugging processes and swift microprocessor functions. As a virtual machine, CUDA is assigned to many streaming multiprocessors, that are arranged as a 32-wide SIMD cores along with multiple thread contexts of more than 512 thread contexts. Thread contexts are majorly bundled into warps of 32 threads each, which are multiplexed onto the SIMD hardware.

On a 2D grid the kernels are recalled, which is sub-separated into a maximum of thread blocks of a capacity of 64K 3D. Every Single thread block is completely mapped to completion on an arbitrary streaming multiprocessor to be executed. Concurrently, the warps are multiplexed on a cycle-by-cycle granularity based on their readiness to execute. In their execution, threads do so in lockstep while discrepancy is treated by a

masking and branch stack. The streaming multiprocessors are characterized by fast and small software-controlled shared memory by each thread in any communicated thread block.

The CUDA Tesla architecture is only confirmed from NVIDIA, which includes the 8-series, Quadro, and GeForce, GPU's and the computing model of Tesla GPU. After launching the kernel, the hardware scheduler is used to assign every single thread block to a single streaming multiprocessor with enough space for holding the entire thread block. In the event that several thread blocks fitting with a single streaming multiprocessor, it will be executed systematically and concurrently since each thread is allocated a scalar, which enables them to execute arbitrary addresses to the respective codes. Sufficient memory is required to facilitate the execution of commands in the respective order. Presumably, the programs contain a single thread; therefore, it is assumed that it is possible to access all the resources from a centralized location.

Building a Dynamic analyzer for a CUDA system requires the assessment memory locations and the processor speed of the underlying system. The system needs to evaluate the limitations of the host system, the capacity of fighting bugs, and the capacity to detect the correctness of the program. Apparently, a former classmate experiences the challenges of sufficient memory for the execution of single or multiple cases. CUDA Tool makes the developing process of the GPGPU simple. Mistakes the performers make is by compiling the system improperly triggering mistakes and reducing the correctness of the code. The aforementioned technique for automatically instrumenting CUDA programs with literally no programmer input can be achieved systematically through the real-time detection of race conditions and inconsistency in the shared and dedicated memories respectively.

4. The Dynamic Analyzer Architecture

Convert the original CUDA source code to an intermediate representation (from C code to CIL code) by the source-to-source compiler, which is a representation that makes it easy to analyze and manipulate C programs and emit them in a form that resembles the original source. The intermediate representation is transformed and instrumented by the instrumentor, then the instrumentor generates instrumented CUDA source code based on specific rules. The instrumented program is then compiled by the compiler. Finally, the dynamic analyzer executes the executable file and output the list of dynamic errors.
The elements of the analyzer system:
A source-to-source compiler: Convert the original CUDA source code to an intermediate representation (from C code to CIL code).
Instrumentor: generate instrumented CUDA source code based on specific rules. The Instrumentor contains a lexical analyzer known as a scanner. It converts the input program into a sequence of Tokens.
Compiler: to compile the instrumented program and convert it to an executable file to executes it and output the list of dynamic errors.

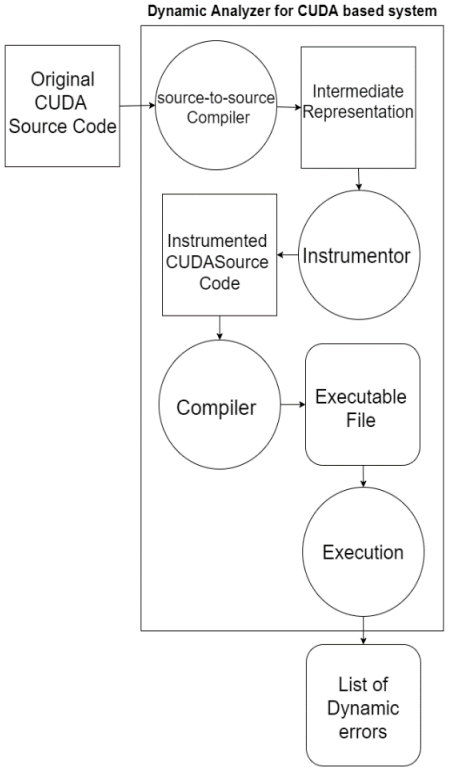

Figure 1: Dynamic Analyzer Architecture for CUDA based system

5. Analyses of CUDA code

As established in the previous sections of the paper, CUDA, the general-purpose application development for GPUs (GPGPU) makes the developing process of general-purpose programs for the GPU simple. According to Kerr, Andrew, and Sudhakar, the general purpose application development for GPUs (GPGPU) provides a cost-effective alternative for accelerating data and computing intensive applications. The current software development aims to enhance effectiveness in the software application process as well as a compiler optimization, and the ultimate transformation of microprocessor processing.

According to Zheng et al., illustrates the cost-effectiveness of GPUs in achieving high performances in image and graphics processing [10]. For this reason, most programmers and media specialists have found favor in the

use of GPUs as opposed to contemporary CPUs. CUDA and OpenCL are some of the programs that have dynamically embraced GPUs for both graphical and non-graphical operations. The desire of most programmers is to achieve the correctness of their programs. Apparently, the GPU micro processing framework offers a multithreaded environment which triggers the occurrence of data races. Data races can negatively affect program reliability. In response, different tools have been devised to detect race conditions and eradicate them beforehand.

5.1 Data Races in CUDA

The current approaches to detecting race conditions are limited by the lack of scalability due to the state explosion problem. State explosion occurs when the number of state variables in the system increase and the size of the system state space increase exponentially (Clarke et al. [2]). Another limitation is the reporting of false positives due to the simplified modeling or the presence of non-lock synchronization primitives. Ultimately, the presence of prohibitive runtime and space overhead can hinder the logical flow of commands during the flow of execution. Essentially, Essentially, GRace could be effectively used to detect data races without bogging the system with unnecessary processes.

5.1.2 Avoiding Race Condition

Mainly, in simple programs such as the one illustrated in figure 2 below, race conditions are detected and corrected manually. but this process is infeasible in complex programs.

```
1.  #define N (2048*2048)
2.  #define THREADS_PER_BLOCK 512
3.  __global__ void dot( int *a, int *b, int *c )
    {
4.  __shared__ int temp[THREADS_PER_BLOCK];
5.  int index = threadIdx.x + blockIdx.x *
    blockDim.x;
6.  temp[threadIdx.x] = a[index] * b[index];
7.  __syncthreads();
8.  if( 0 == threadIdx.x ) {
9.  int sum = 0;
10. for( int i = 0; i < THREADS_PER_BLOCK; i++ )
11. sum += temp[i];
12. *c += sum;
13. atomicAdd( c , sum );
14. }
15. }
```

Figure 2: CUDA program with Race Condition

The Race Condition error happen if two or more threads need to access and operate on a memory location without synchronization, as shown below in figure 3.

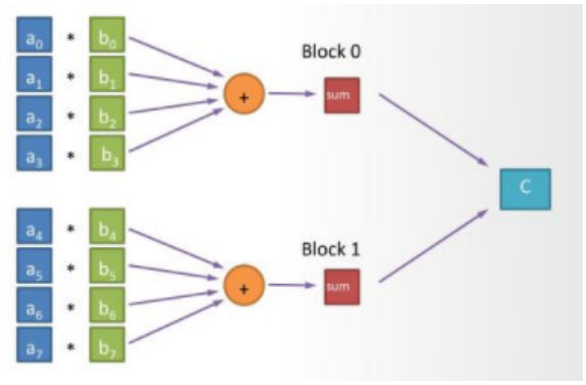

Figure 3: CUDA program with Race Condition

The source code of the CUDA program is transferred as an intermediate representation to recalls declarations knowledge and type qualifiers for the CUDA-specific. By using the C Intermediate Language framework. Then the analyzer is instrumented and transformed the intermediate representation. These transformations are just to use these transformations by global or device and shared functions, after that the analyzer converts the instrumented representation to Cuda's dialect of C.

GRace utilizes analysis to reduce the number of statements that are entailed in the procedural instrumentation [10]. Thread scheduling and the execution models are significant in the detection of data races without generating reports on false positives. The schemes, GRace-stmt and GRace-addr are specifically designed for NVIDIA GPUs. They are purposely designed for dynamic evaluation, logging, and analysis during runtime. They are also integrated using the same scheme analysis which simplifies the algorithmic development. They are also integrated with three data race bugs in three GPU kernel functions, which is critical in the analysis and articulation of the critical GPU processes.

The Btool is established as a secondary mechanism to report false positives [10]. Apparently, GRace-addr incurs a low runtime overhead and a low space overhead after a comprehensive analysis on the kernels. Grace-stmt, on the other hand, offers diagnostic procedures on all matters involving data races that exceed the threshold overhead. The two schemes have been used to transform the programmability of GPUs, leading to nearly a perfect micro processing mechanism using GPUs. The schemes have also enhanced the utilization of GPUs in multiple capacities that require extensive data processing. A wide variety of users with little programming experience can scale their applications using GPUs with overwhelming ease.

Referring to the CUDA code illustrated in figure 2, it can be noted that a race condition can be seen on line 12 of the code. After running the code through an automated instrumentation tool, Evidently, automated instrumentation

detects and reports race conditions comprehensively. It further adds a call to syncthreads between lines 11 and 12; however, re-running the code does not report a race condition.

5.2 Performance

Understanding the performance of the CUDA program can significantly influence the implementation dynamics involving the appropriate resource allocation in terms of memory and processing speed. In some cases, developers parallelize various applications with the aim of optimizing performance and enhancing memory utilization. Apparently, parallelizing the application would else not be applied if it negatively influences the performance of the application. Alternately, the application would not be allocated dedicated resources if its performance cannot be significantly verified.

Parallelizing can be critical if attributed to some significant improvements in the overall improvement of the application. Therefore, maximizing performance on a CUDA program entails the utilization of share memory, enhancement of efficient debugging, and advanced micro processing using the GPU.

According to Fang, Ana and Henk, although CUDA and OpenCL are critical in image processing, there is a significant difference in their performance [3]. Differences are exhibited in the programming models used to enhance their mode of operation, optimization strategies, background compilers, and the architectural framework. Inherently, CUDA is far more efficient than OpenCL; however, to basic computer users, the difference is insignificant. Additionally, how effective each of the performance depends on the secondary features such as the processing capacity of the computing devices. OpenCL is also widely used as an alternative to CUDA or in the conduction of analytical reports on the performance of both applications.

Che et al. provides that GPUs are characterized by simplistic, data-parallel, deeply multithreaded cores, and high-memory bandwidths, which are easily programmable and effective in advanced image processing [1]. As aforementioned, speedups for a variety of general-purpose applications make GPUs incomparable to the contemporary general-purpose processors (CPUs) in terms of speed and portability. The C-like programming language used by NVIDIA has enhanced the design of highly effective image processors. In addition to the programming language, programmers have established specific coding idioms to enhance GPU performance under extreme system utility. The ultimate goal of these approaches is to ensure that the GPU is to enhance performance and meet the growing demand in the current market where image processing is largely widespread.

Farooqui et al. [4] proposed the design and implementation of a dynamic instrumentation infrastructure for PTX programs that procedurally transform kernels and manage related data structures. These can be achieved by the use of a GPU Ocelot dynamic compiler infrastructure which is only unique to PTX programs. The profiling and instrumentation tool chains facilitate the acceleration of the workload being executed by the GPU, provide information that can be used to manage load imbalance, and enhance the utilization of the resource allocator to enhance performance.

Ultimately, these too chains facilitate the compute utilization feedback which is enhanced by the simulated process scheduler. The hypervisor is critical in the establishment of a feedback platform, where the user or programmer can utilize real-time information to manage system resources for ultimate microprocessor performance. Occasionally, compilation overheads may be necessary while performing dynamic compilation during a series of GPU processes. These activities increase runtimes during the execution of instrumental kernels in extensive processing frameworks. According to Farooqui et al [4], compilation overheads constitute 69% of the time required to complete the execution as a result of instrumentation modules on the kernel. This phenomenon provides a critical framework within which the system can be optimized using the readily available features on the system.

5.2.1 Bank Conflicts

Accessing global memory is enhanced by system developers due to the fact that such shared memory is located on a chip. Mostly, for easy accessing of global memory, the system is designed such that it cannot be cached. Consequently, the process facilitates fast access to shared memory, which obviously depends on the accessibility pattern. According to Harris, the shared global memory is split into 16 banks which hold sequential words in memory [11]. The access speed is significantly high following the ability of the threads to access the same bank. In this case, the speedy access can be effectively equated to register access.

The access of a bank by a series of registers enhances the serialization of the accesses of the respective threads, which decreases the aggregate throughput of the CUDA program (Ruetsch et al. [13]). The banking strategy is strategized to enhance the scalability of the memory being accessed by the respective users of the host program. Whenever different threads access the memory, they are serialized to enhance a systemic development approach in the articulation of the accessibility framework. The number of cores in the multicore processors increases

gradually in line with the increase in the need for scalability.

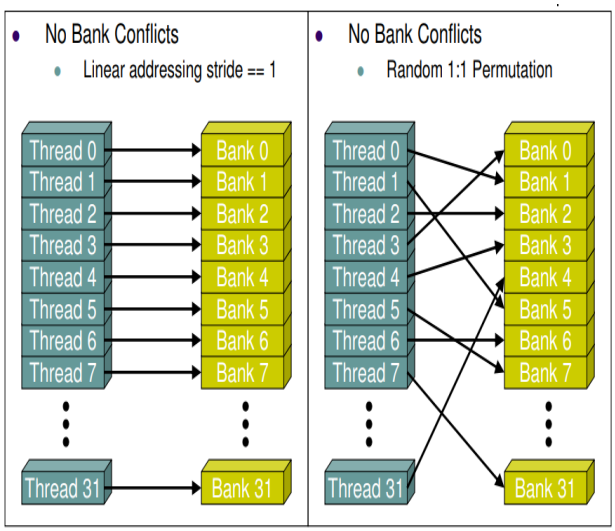

Figure 5: Bank Conflicts

The performance of the program when the bank conflicts occur if four times the performance of the program without any conflicts. Apparently, such performances correspond to worse than if the program was rewritten the slow, off-chip global memory (Iwai et al. [14]). Evidently, in order to improve the performance and optimization of CUDA programs, the developer must significantly reduce any occurrence of such conflicts.

5.2.2 Avoiding Bank Conflict

The initial step towards reducing bank conflicts is being able to detect them. The analysis of memory accesses is used to detect bank conflicts in simple programs. The correctness analysis approach, which uses a manual analysis approach can also be used to detect bank conflicts; however, the approach becomes infeasible when the programs become complex. The use of automatic bank detection is thus mostly used to address the challenges of bank conflicts (Ryoo et al. [12]). The use of a highly particular instrumentation code is inserted into the second step to detect bank conflicts immediately they start to build up.

Additionally, the use of a global array enhances the storage of the addresses accessed by each thread in the program. A code is always added for each thread so that it can be able to update its entry in the array every time the shared memory is accessed. A synchronism between the memory access and code integration enables the threads to calculate each access to memory bank by all threads, which determines if there are bank conflicts and determines the impact of these bank conflicts. All information regarding the access and the presence of bank conflicts is progressively displayed to the program developer.

6. Testing Results

The use of an explicit synchronization can be used to avoid race conditions, which facilitate the definition of a specific macro that can be used to avoid bank conflicts. Illustratively, removing the synchronization statements in the program can be used to observe the performances of the tools while detecting the race conditions that arise in the course of the implementation process. In addition to observing the race conditions, the approach can enhance the detection of the bank conflicts as well as the detection of the resultant lack of bank conflicts.

6.1 Analysis of CUDA code Correctness

The race condition analysis was illustrated scan and the scan performance of the program analyzed using syncthreads calls. The analysis of the unmodified version indicated no race conditions. Modification of the three versions of scan and removal of the syncthreads reported race conditions following the removal of the synchronization point.

6.2 Analysis of CUDA code Performance

The performance was conducted by the bank conflict analysis using the original version of the scan that had been redesigned to observe the performance of the program with or without the bank conflicts. The performance analysis enabled the developers to analyze the severity of having bank conflicts and approaches that could be used to eradicate them effectively. Ultimately, the illustrations indicated that the user could effectively develop an automated analysis to check the program development while the user code without the fear that bank conflicts could lower program performance.

6.3 Performance Impact of Instrumentation

Running the CUDA applications in emulation mode is mostly recommended due to its improved debugging capabilities. Primarily, an instrumented source code produced by the source code presented earlier in the study provides that it is advisable to run the code in device emulation mode as opposed to the usual GPU hardware. Majorly, comparisons between the runtime of the original application and the instrumented code running in the

simulation code can be used to determine the computing and performance overhead of the instrumentation code.

7. Related Work

Several dynamic analyses frameworks are used to detect and address concurrency errors in writing CUDA programs. The Eraser tool is one of the common systems that transform programs to track locks held on various programs [15]. Following the occurrence of syncthreads; however, memory locks are applied in streamlining the integrity of the CUDA program. Parallel programs can also be applied in addressing the issues associated with the development of efficiency in the coding process.

Yang and Huiyang observed that parallel programs comprise a series of code sections, which are characterized by a diver thread-level parallelism (TLP) [8]. Therefore, a thread occurring in a parallel program such as a GPU kernel in CUDA programs contains parallel loops and sequential codes. NVIDIA utilizes dynamic parallelism to enable a GPU thread to initiate another GPU kernel thus reducing the overhead of launching kernels from the contemporary central processing unit (CPU). Yang and Huiyang further propose the use of control flow in activating different numbers of threads on various code sections [8]. The nested parallelism in CUDA can be implemented utilizing a directive-based compiler method. Optimized GPU kernels are automatically generated once the CUDA-NP is initiated. Consequently, such optimized GPU kernels enhance the efficient management of on-chip resources while reducing scan primitives. CUDA-NP has been used to improve the overall GPU performance by 6.69 times. Michael et al. [16] built a dynamic analyzer to detect the CUDA code errors, which enhances the performance of the CUDA programs.

8. CONCLUSION

Characteristically, a parent thread can be able to communicate with the child threads; however, it can only do so via global memory. Additionally, the overhead of the parent graphics processing unit GPU lacks any triviality even within the GPUs. Nonetheless, it is possible to show that the existing GPGPU benchmarks containing parallel loops have a relatively low loop count. Occasionally, these benchmarks are characterized by high degrees of thread-level parallelism. In this case, therefore, leveraging such parallel loops using dynamic parallelism can be unable to offset its overhead, making it difficult to utilize the GPU processing speed and system resources. The use of nested parallelism in CUDA may increase the number of threads when initiating the GPU program.

References

[1] Che, Shuai, et al. "A performance study of general-purpose applications on graphics processors using CUDA." Journal of parallel and distributed computing 68.10 (2008): 1370-1380.

[2] Clarke, Edmund, et al. "Progress on the state explosion problem in model checking." Informatics. Springer, Berlin, Heidelberg, 2001.

[3] Fang, Jianbin, Ana Lucia Varbanescu, and Henk Sips. "A comprehensive performance comparison of CUDA and OpenCL." Parallel Processing (ICPP), 2011 International Conference on. IEEE, 2011.

[4] Farooqui, Naila, et al. "A framework for dynamically instrumenting GPU compute applications within GPU Ocelot." Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. ACM, 2011.

[5] Kerr, Andrew, Gregory Diamos, and Sudhakar Yalamanchili. "A characterization and analysis of ptx kernels." Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, 2009.

[6] Li, Peng, Guodong Li, and Ganesh Gopalakrishnan. "Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs." Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, 2012.

[7] Sanders, Jason, and Edward Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional, 2010.

[8] Yang, Yi, and Huiyang Zhou. "CUDA-NP: realizing nested thread-level parallelism in GPGPU applications." ACM SIGPLAN Notices. Vol. 49. No. 8. ACM, 2014.

[9] Yang, Zhiyi, Yating Zhu, and Yong Pu. "Parallel image processing based on CUDA." Computer Science and Software Engineering, 2008 International Conference on. Vol. 3. IEEE, 2008.

[10] Zheng, Mai, et al. "GRace: a low-overhead mechanism for detecting data races in GPU programs." ACM SIGPLAN Notices. Vol. 46. No. 8. ACM, 2011.

[11] Harris, Mark. "Optimizing cuda." SC07: High Performance Computing With CUDA, 2007.

[12] Ryoo, Shane, et al. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA." Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM, 2008.

[13] Ruetsch, Greg, and Paulius Micikevicius. "Optimizing matrix transpose in CUDA." Nvidia CUDA SDK Application Note 18 (2009).

[14] Iwai, Keisuke, Takakazu Kurokawa, and Naoki Nisikawa. "AES encryption implementation on CUDA GPU and its analysis." Networking and Computing (ICNC), 2010 First International Conference on. IEEE, 2010.

[15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. "Eraser: A dynamic data race detector for multithreaded programs". ACM Transactions on Computer Systems, 15(4):391–411, 1997.

[16] Michael Boyer , Kevin Skadron , Westley Weimer "Automated Dynamic Analysis of CUDA Programs"

[17] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in International Symposium on Software Testing and Analysis. ACM, pp. 19–30, 2019.

Salah T. Alshammari: Ph.D. student at king abdulaziz university, department of computer science, college of computing and information technology, Jeddah, Saudi Arabia. His main research interests are Information Security, Cybersecurity, Security in Cloud Computing, Trust in Cloud Computing, Software Testing.