International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 14 (2017) pp. 4282-4285
© Research India Publications. http://www.ripublication.com

Visual Analytics of Millions of GPU Threads

SeongKi Kim[1,*] **and HyukSoo Han**[2]

Sangmyung University,
20, Hongjimun 2-gil, Jongno-gu, Seoul, Republic of Korea
**Corresponding author*

[1]*Orcid: 0000-0002-2664-3632,* [2]*Orcid: 0000-0003-2369-7202*

Abstract

Although the GPGPU has been widely used in various fields for algorithms acceleration, it is notorious for its programming difficulties because of many different concepts from general CPU programming and issues from the huge number of concurrent threads. In order to verify GPGPU software, we have developed a visualization tool known as GPGPU-Vis, which shows the run-time actions of all threads. However, it is still challenging to visualize millions of threads, due to time and memory limitations. To successfully visualize them, we propose reduction methods for threads containing duplicated information. Our methods can reduce the number of threads from millions to only one in many cases, and we can effectively visualize GPGPU software within given resource limitations. Through the suggested reduction methods, our tool can visualize the millions of threads with a minimum information loss, can help verifying the GPGPU software.

Keywords: Visualization, GPGPU, GPU, Thread, Verification

INTRODUCTION

With GPU performance improvements and the advent of GPGPU, GPGPU programming has recently been widely applied in various fields. In artificial intelligence, the GPU has been used to accelerate deep learning [1] and machine learning [2]. In the graphics field, the GPU has been used for collision culling among millions of objects [3], as well as intersection tests among the ray and objects [4]. Furthermore, the GPU has been applied to supercomputing, and many GPU-based supercomputers are listed in the Top500 [5].

However, GPGPU programming is notorious for its difficulties in the aspects of logic and intervention. In terms of logic, the GPU threads exhibit different characteristics from those of the CPU, in that the GPU threads in the same group run the same code by means of a lockstep method. A further difference in the logic is that communication through memory is not guaranteed, because inter-group memory coherency is not supported. The number of threads and group size are also different to those of general CPU threads. When running GPU threads, the number and size should be specified, as these affect software behavior.

Moreover, interventions among threads can cause many problems, an example of which is the data race. This occurs when two or more threads access the same memory location, and one of these is write access. The second intervention example is the barrier divergence that occurs when certain threads run a barrier function and other threads do not execute this function. The results of barrier divergence are unexpected from vendors to vendors, due to no mention of this in the specification. Deadlock as a result of the lockstep execution is the third intervention issue, which occurs when certain threads in a group can exit a loop, but others cannot. In this case, no threads are able to exit the loop due to the GPU's lockstep execution.

In order to alleviate these difficulties for developers and aid in verifying GPGPU software, we developed the GPGPU-Vis, which is a visualization tool, and evaluated it using a variety of GPGPU software. This tool can assist in analyzing the actions of each thread, as well as detecting the data race, barrier divergence, and deadlock. However, when attempting to visualize software with millions of threads, the tool requires a long time or fails, because it must check every instance of memory access and monitor every executed instruction (millions of threads are common in a lot of GPGPU software). Moreover, when an HTML browser shows developers the visualization results, it also requires a great deal of time because the browser needs to parse the generated HTML files

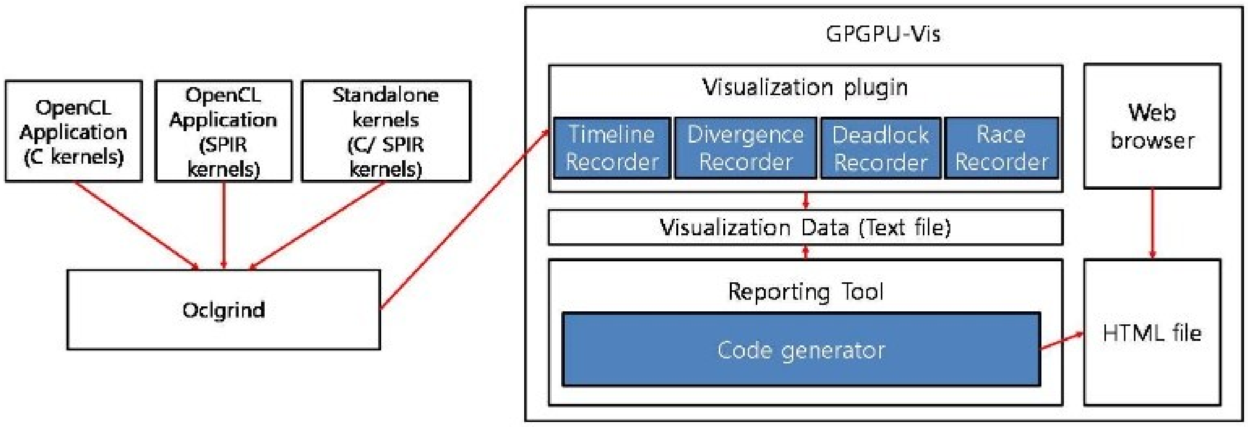


Figure 1. Overall architecture of GPGPU-Vis.

by millions of threads, and display the results on a webpage.

This paper makes the following contributions to GPGPU software research. Firstly, we propose a novel method for reducing the number of threads requiring additional analysis. Secondly, we propose a novel method for reducing the number of visualizing threads with duplicated information. Thirdly, we successfully visualize the kernels (GPGPU codes executed) with millions of threads by means of the proposed methods. The remainder of this paper is organized as follows. Section 2 describes the GPGPU-Vis, which is a visualization tool of GPU threads. Section 3 describes our thread reduction methods and demonstrates the results, and our conclusion is provided in section 4.

GPGPU-Vis

The GPGPU-Vis is a visualization tool for GPGPU software, and its overall architecture is illustrated in Fig. 1.

In Fig. 1, it can be seen that the GPGPU-Vis is largely composed of two subcomponents: a visualization plugin and reporting tool. When the Oclgrind [6] runs GPGPU software with a plugin option, it calls the plugin, which records a timeline, data race, barrier divergence, and deadlock, and then generates visualization data including the detected information. The reporting tool reads the visualization data and generates an HTML file that uses the vis.js [7] library for visualization. Once the reporting tool has created the HTML file, the results can be observed through any HTML browser.

In our experiments, the architecture shown in Fig. 1 can successfully visualize results if the number of threads is less than 256 and the analyzed kernel is simple. However, when the number is more than 256 or the kernel is complex, the analysis is lengthy or fails due to memory limitations. Even if the tool successfully visualizes the results, it is challenging to investigate 256 or more threads in an HTML page, and investigations are error-prone.

However, a great deal of GPGPU software contains millions of threads and hundreds of lines. Table 1 below displays the number of threads and the lines of code in the Rodinia benchmark [8]. In Table 1, we include the maximum number of threads if the software calls several kernels.

Table 1 indicates that the software uses a large number of threads and OpenCL kernels with lengthy lines of code. In particular, the numbers of threads and groups are ten of millions and tens of thousands, respectively, in the pathfinder case, and it is therefore difficult to visualize all of the threads. Even if we can visualize all of the threads, it is still challenging to investigate all of the actions of each thread. This paper describes methods that can simplify the enormous number of threads and the resulting displayed threads. In Table 1, ∞ indicates that the software does not specify the number of threads in a group, and in this case, the OpenCL implementation internally determines the group size.

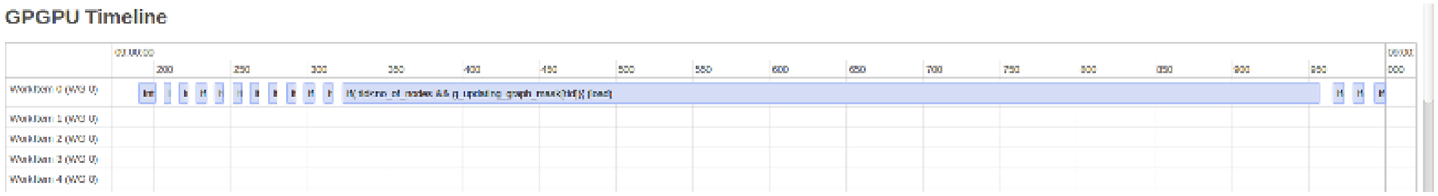
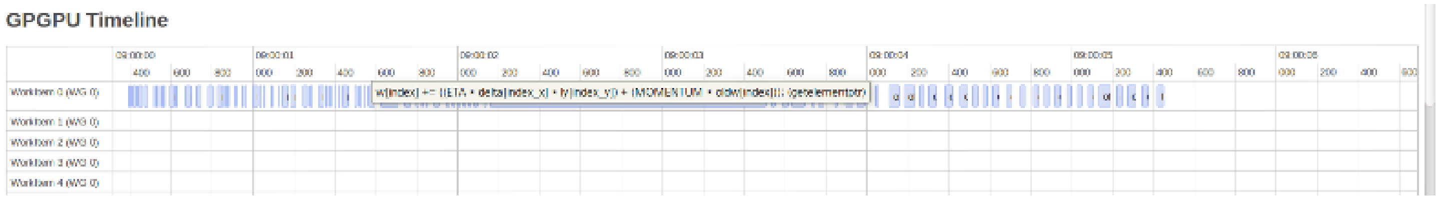


Figure 2. Visualization of bfs software.



```
w[index] += (ETA * delta[index_x] * ly[index_y]) + (MOMENTUM * oldw[index]); (getelementptr)
```

Figure 3. Visualization of backprop software.

Table 1. Numbers of threads and groups.

Name	Number of Threads	Number of Groups	Line of codes
b+tree	2560000	10000	220
backprop	1048576	4096	90
bfs	1000192	3907	50
cfd	388608	2024	284
dwt2d	1048576	4096	707
gaussian	65536	∞	49
heartwall	13056	51	2235
hotspot	473344	1849	115
hotspot3D	262144	1024	50
kmeans	494080	1930	61
lavaMD	128000	1000	284
leukocyte	104448	408	581
lud	1016064	3969	162
myocyte	4	2	1445
nn	42816	∞	21
nw	2048	128	202
particlefilter	400384	782	752
pathfinder	10000000	10000	116
srad	230144	899	346
streamcluster	3145728	12288	68

THREAD REDUCTION

In order to reduce the results of the GPGPU-Vis described in section 2 with minimal information loss, we use two-level approaches for the analysis and visualization aspects.

Reduction of Analyzed Threads

In GPGPU programming, global synchronization among different groups is not guaranteed. As a result, GPU developers write code that each group runs independently, without any inter-group communication. Because of this separated characteristic of GPGPU programming, we decrease the number of groups to only one, in order to reduce the analyzed threads, because all of the groups run the same GPGPU kernel. This means that the number of analyzed threads can be reduced to the number of threads in a given group. If the number is zero (for example, as in the gaussian and nn cases in Table 1), we use the size internally determined by the OpenCL implementation.

Reduction of Visualized Threads

Even if the number of analyzed threads can be reduced, certain software still includes many threads that need to be visualized. In particular, the pathfinder contains a complex kernel and a thousand threads within a single group. Due to time and memory limitations, it is still challenging to visualize all of the threads. Therefore, we also remove duplicated information if a thread runs the same instruction sequences as the previous ones. For example, if thread 2 runs the same series as thread 1, thread 2 is not displayed.

Implementation and Results

When implementing the reductions described in this section, we modified the tested software and GPGPU-Vis. In the tested software, we adjusted the number of threads to the group size, so that only one thread group is run. We furthermore modified the GPGPU-Vis to generate an HTML file that skips the instructions if a thread runs the same instructions as the previous thread. When running the software displayed in Table 1, only the bfs and backprop were successfully executed on the Oclgrind; therefore, we used this software for our reduction verification. The bfs and backprop with millions of threads are displayed in Fig. 2 and Fig. 3.

In the bfs and backprop cases, the thread numbers are 1000192 and 1048576, respectively. However, we decrease these numbers to 256 by reduction of analyzed threads and 1 by reduction of visualized threads, respectively.

CONCLUSIONS

It is known to be difficult to develop GPGPU code because of different behaviors from general CPU code. To relieve the burden of GPGPU researchers and developers, we developed the GPGPU-Vis, which can visualize run-time actions.

However, when attempting to use the GPGPU-Vis in a variety of GPGPU software, a great deal of time is required, and it is at times impossible to store all instances of memory access. Furthermore, it is challenging to investigate all of the threads, because the thread number is more than a million and computer resources have limitations.

In order to address this issue without any information loss, we reduce the number of analyzed threads and visualized threads. We applied the general concepts that each group independently runs the same code without any inter-communication, and most threads execute the same series of code. Through the reduction of duplicated information, we can successfully visualize GPGPU programs with millions of threads. To the best of our knowledge, this paper provides the first research to realize the visualization of millions of threads in the GPU.

ACKNOWLEDGEMENT

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (2015-0-00445) supervised by the IITP (Institute for Information & communications Technology Promotion), and NRF in Korea (2015R1C1A1A01051839).

REFERENCES

[1] H. Cui, H. Zhang, G.R. Ganger, P.B. Gibbons, E.P. Xing, GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server, in: Proceedings of the Eleventh European Conference on Computer Systems, EuroSys ’16, ACM, New York, NY, USA, 2016, pp. 1-16.

[2] J. Jia, P. Kalipatnapu, Y. Yang, Building a Distributed, GPU-based Machine Learning Library, Tech. Rep. UCB/EECS-2016-112, EECS Department, University of California, Berkeley (May 2016).

[3] F. Liu, T. Harada, Y. Lee, Y.J. Kim, Real-time Collision Culling of a Million Bodies on Graphics Processing Units, ACM Trans. Graph. 29 (6) (2010) 154:1-154:8.

[4] V. Shumskiy, Transactions on computational science xix, Springer-Verlag, Berlin, Heidelberg, 2013, Ch. GPU Ray Tracing: Comparative Study on Ray-triangle Intersection Algorithms, pp. 78-91.

[5] Top500, Top 500 the list. @ONLINE (Mar. 2017). URLhttps://www.top500.org/lists/2016/11/

[6] J. Price, S. McIntosh-Smith, Oclgrind: An Extensible OpenCL Device Simulator, in: Proceedings of the 3rd International Workshop on OpenCL, IWOCL '15, ACM, New York, NY, USA, 2015, pp. 12:1–12:7.

[7] vis.js. @ONLINE (Mar. 2017). URLhttp://visjs.org/

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A Benchmark Suite for Heterogeneous Computing, in: Proceedings of IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 44-54.