applied sciences

MDPI

Article

Performance Analysis of Thread Block Schedulers in GPGPU and Its Implications

KyungWoon Cho [1] and Hyokyung Bahn [2,*]

1 Embedded Software Research Center, Ewha University, Seoul 03760, Korea; cezanne@ewha.ac.kr
2 Department of Computer Engineering, Ewha University, Seoul 03760, Korea
* Correspondence: bahn@ewha.ac.kr; Tel.: +82-2-3277-4247

Received: 30 November 2020; Accepted: 17 December 2020; Published: 20 December 2020

Abstract: GPGPU (General-Purpose Graphics Processing Unit) consists of hardware resources that can execute tens of thousands of threads simultaneously. However, in reality, the parallelism is limited as resource allocation is performed by the base unit called thread block, which is not managed judiciously in the current GPGPU systems. To schedule threads in GPGPU, a specialized hardware scheduler allocates thread blocks to the computing unit called SM (Stream Multiprocessors) in a Round-Robin manner. Although scheduling in hardware is simple and fast, we observe that the Round-Robin scheduling is not efficient in GPGPU, as it does not consider the workload characteristics of threads and the resource balance among SMs. In this article, we present a new thread block scheduling model that has the ability of analyzing and quantifying the performances of thread block scheduling. We implement our model as a GPGPU scheduling simulator and show that the conventional thread block scheduling provided in GPGPU hardware does not perform well as the workload becomes heavy. Specifically, we observe that the performance degradation of Round-Robin can be eliminated by adopting DFA (Depth First Allocation), which is simple but scalable. Moreover, as our simulator consists of modular forms based on the framework and we publicly open it for other researchers to use, various scheduling policies can be incorporated into our simulator for evaluating the performance of GPGPU schedulers.

Keywords: thread block; GPGPU; thread block scheduling; Round-Robin

1. Introduction

With the rapid advances in many-core hardware technologies, GPGPU (General-Purpose GPU) has expanded its area from graphical processing to various parallel processing jobs. Specifically, as the era of the 4th Industrial Revolution emerges, GPGPU has become an essential computing device in various domains such as deep learning, block-chain, and genome analysis [1,2]. GPGPU has tens of thousands of computing units, which can maximize their performances by executing threads in each unit on parallel [3]. In general, total threads to be executed are grouped into a certain number of threads called thread blocks, which are the allocation unit for hardware resources. For example, in CUDA [4,5], which is a representative GPGPU programming platform, each thread is included in a thread block, which is identified by the three-dimensional indices, and the thread blocks can be represented as a three-dimensional grid to define a GPGPU task as shown in Figure 1. Meanwhile, since GPGPU hardware is composed of tens of stream multiprocessors (SM), a thread block scheduler that allocates thread blocks to SM is necessary. In the current system architecture, the thread block scheduler is implemented as hardware in the GPGPU device.

Figure 2 shows the basic role of the thread block scheduler, which is to track the resource demands of thread blocks allocated to each SM and perform scheduling so that the allocated thread blocks do not exceed the given resources in SM. The type and amount of resources in SM required to perform a single thread block can be statically determined while building a GPGPU task. Thus, it is not complicated to perform scheduling so as not to exceed the resource capacity of SM. However, as the number of threads allocated in SM increases, the execution time increases dramatically, and thus it is essential to allocate thread blocks to SM in a balanced manner [6,7].

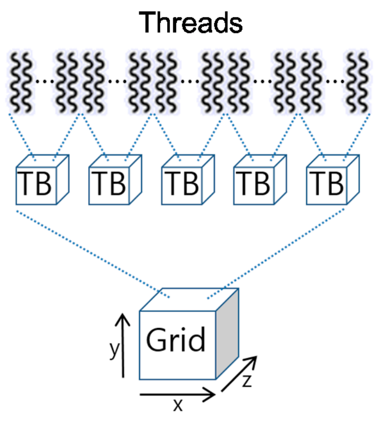

Figure 1. Grouping a certain number of threads to TB (Thread Blocks) in a GPGPU task.

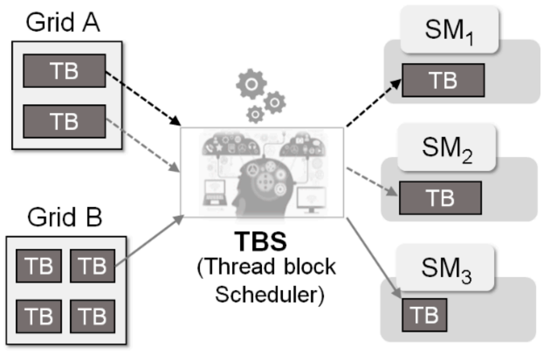

Figure 2. An example of Thread Block Scheduling (TBS) that allocates Thread Blocks (TB) to Stream Multiprocessors (SM).

To schedule thread blocks in GPGPU, the current hardware scheduler allocates thread blocks to SM by making use of the Round-Robin scheduling algorithm. Round-Robin is a resource allocation algorithm widely used in CPU scheduling [8,9] and network packet transmission [10]. As Round-Robin scheduling is simple, easy to implement, and starvation-free, it is efficient to implement in hardware [11]. However, we observe that Round-Robin is not efficient in GPGPU as it does not consider the workload characteristics of threads and the resource balance among SMs. In this article, we present a new thread block scheduling model that has the ability of analyzing and quantifying the performances of thread block scheduling. We implement our model as a GPGPU scheduling simulator and show that the conventional thread block scheduler provided in GPGPU hardware does not perform well as the workload becomes heavy. Specifically, we observe that the performance degradation of Round-Robin can be eliminated by adopting DFA (Depth First Allocation), which is simple but scalable. As our simulator consists of a modular form based on the framework and it is publicly available [12], various scheduling policies can be incorporated into our simulator for evaluating the performance of GPGPU schedulers.

The remainder of this article is organized as follows. Section 2 describes the GPGPU structure and the thread block model. Section 3 explains our GPGPU simulator for thread block scheduling. In Section 4, we perform experiments and present the performance comparison results on scheduling policies with the current system and two new policies. Finally, Section 5 concludes this article.

2. GPGPU and Thread Block Model

As SM adopts the SIMT (Single Instruction Multiple Thread) model, it executes the same instruction for multiple threads simultaneously [13]. The maximum number of threads that can be executed per SM is generally 2048. The thread block scheduler should manage the allowable number of threads per SM not to exceed this limit while allocating thread blocks to SM [14]. Meanwhile, a series of threads that should be executed simultaneously within the same hardware unit are called *Warp*, which can execute up to 32 threads simultaneously. Each thread block consists of at least one Warp, and whenever the number of threads constituting a Warp exceeds 32, the number of Warps increases by 1. The number of Warps that can be executed at the same time depends on the number of each computing unit in SM [15]. From this point of view, SM does not execute all threads in the thread blocks simultaneously, but Warps for the given thread block are allocated to the computing units and then the threads in that Warp are executed simultaneously.

When scheduling thread blocks, not only the number of threads per SM but also the number of Warps should be considered. That is, when allocating thread blocks to SM, the thread block scheduler should monitor the utilization of each type of resources in SM and manage them so that SM does not exceed allowable resource usage. Specifically, when performing thread block scheduling, the following types of resources should be monitored.

- Registers and shared memory
- Threads and thread block Information
- Hardware Warp information

Thread blocks belonging to the same GPGPU task have the same resource demands, but the resource requirements are varied as the thread blocks are from different tasks. Due to this reason, the thread block scheduler should monitor and manage the usage of each resource type in each SM. For example, even though the resource utilization of SM is not high, if the resource type to be used is already in use, Warps that have not yet been assigned the resource should wait. This is called *Stalled Warp*, which is the main reason of GPGPU's performance degradations. In addition to stalls by computing units, stalls due to memory references also cause serious performance degradation.

When the shared memory is used between threads belonging to the same thread block, the delay is up to 100 cycles as the number of threads increases. In the case of global memory sharing across threads in different SMs, the delay becomes hundreds of cycles or more [16–18]. The two types of memory have a tradeoff between latency and capacity so those should be deliberately utilized considering data input size and copy overhead [19,20].

3. A GPGPU Simulator Based on the Thread Block Scheduling Model

In order to simulate the thread block scheduling function in GPGPU, an accurate modeling of the hardware resources and tasks in GPGPU is required [21]. For quantifying the impact of the thread block scheduling policy on GPGPU task performances, we model SM and GPGPU tasks based on thread blocks. Specifically, in the proposed model, the available resources for each thread block in SM are managed based on the micro thread block (mTB), which is up to 32 within a Warp.

Figure 3 shows the measured execution time on the GPGPU platform we experimented when the total number of threads to be executed is fixed to 1024 but the number of thread blocks and the number of threads within a thread block are varied. As shown in the figure, the total execution time is not related to the size of a thread block, implying that modeling of thread blocks and SM resources by the unit of mTB can reflect the actual performances well. In our simulation model, tasks can arrive during the simulation period, and the thread block scheduler performs scheduling in the order of their arrival. A GPGPU task for thread block scheduling can be defined by the following attributes.

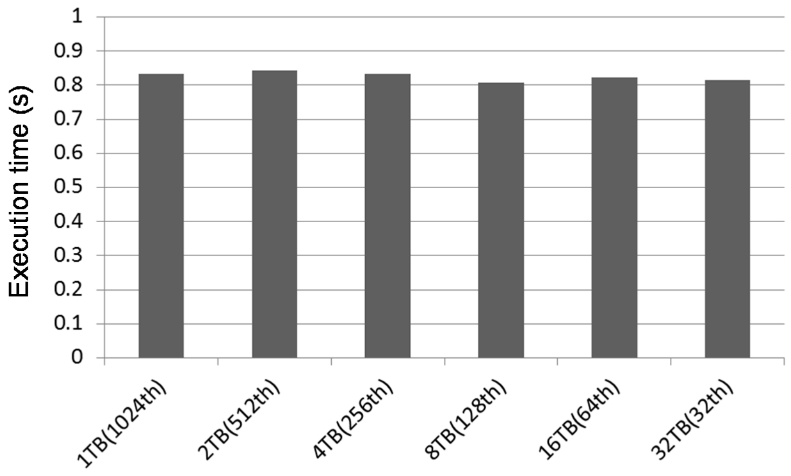

Figure 3. Execution time as the thread block size is varied with the same number of total threads; 1 TB (1024th) = 1 thread block consisting of 1024 threads; 2 TB (512th) = 2 thread blocks with each thread block consisting of 512 threads; 4 TB (128th) = 4 thread blocks with each thread block consisting of 256 threads; 8 TB (128th) = 8 thread blocks with each thread block consisting of 128 threads; 16 TB (64th) = 16 thread blocks with each thread block consisting of 64 threads; 32 TB (32th) = 32 thread blocks with each thread block consisting of 32 threads.

– Arrival time of the task
– Number of mTBs corresponding to the total number of threads
– Computing resource demands per mTB
– Memory resource demands per mTB
– Basic execution time of mTB

Note that the basic execution time of mTB is the time required to complete the mTB if it is the only thread block allocated to the SM. The execution time actually increases due to the resource contention with other mTBs allocated to the same SM [22,23]. A GPGPU task consists of a number of mTBs, and the resources required for executing mTB consist of computing resources and memory resources. In our model, the usage of the computing and memory resources is evaluated based on mTB. That is, our model keeps track of the available computing resources in each SM, and when mTB is allocated to SM, the remaining resources are recalculated based on the resource usage of the mTB. If the remaining resource in SM is not sufficient for the mTB, scheduling to that SM is not possible. Our model also keeps track of the global memory resources in GPGPU, and the memory resource usage per mTB in all SMs are calculated to maintain the remaining memory resources. When mTB is allocated to a certain SM, the remaining memory resource is recalculated. If the remaining memory resource is not sufficient, scheduling of mTB is not allowed.

The scheduler model we propose provides the function of calibrating the basic execution time in order to reasonably configure the task's execution time according to the load of GPGPU. By identifying the load of computing and memory resources according to the computing amount and the global memory references in each SM, respectively, our model estimates the actual execution time of mTB.

The current GPGPU makes use of the Round-Robin scheduling policy that sequentially allocates th read blocks to SM. Although this is simple to implement, it may cause the bias of loads among SMs [17]. In this article, we use two scheduling policies called BFA (Breadth First Allocation) and DFA (Depth First Allocation) as shown in Figure 4. BFA monitors the resource utilization in each SM and allocates thread blocks preferentially to the SM that has the lowest load. By doing so, the resource utilization among SMs can be balanced, which eventually leads to the reduction of the execution time. Unlike BFA, DFA allocates thread blocks to SMs with the highest resource utilization. This is a scheduling policy that can increase the utilization of GPGPU if the load becomes high. Figure 5 shows an example situation that can be accommodated by DFA, which is not possible if we apply BFA. We implement our GPGPU simulator based on the proposed thread block scheduler model. We have opened our simulator for other researchers to replay it [12].

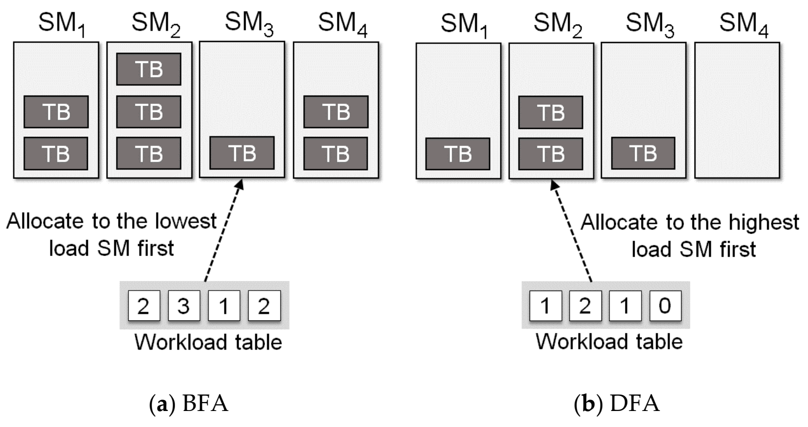
SM1
SM2
SM3
SM4
TB
Allocate to the lowest load SM first
2 3 1 2
Workload table
(a) BFA
Allocate to the highest load SM first
1 2 1 0
Workload table
(b) DFA

Figure 4. Comparison of the thread block allocation between (**a**) Breadth First Allocation (BFA) and (**b**) Depth First Allocation (DFA).

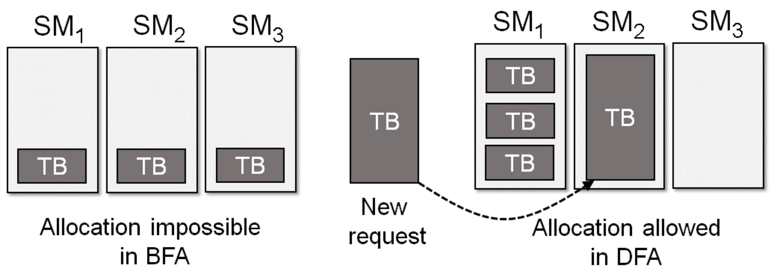

Figure 5. An example situation that can be accommodated by DFA, which is not possible in BFA.

In addition to BFA and DFA, various scheduling policies can be incorporated into our simulator as we made it as a modular form based on the framework. We designed all configuration parameters, such as the number of SMs, computing and memory resources, and workload characteristics to be adjustable, and the simulator also has the function of creating workloads automatically.

4. Performance Evaluation

In order to simulate thread block scheduling, we generate workloads by configuring the resource demand and the execution time of each task based on the desired utilization. The desired utilization here means the ratio of required computing resources for the given workload under the full SM resources in GPGPU. The experimental platform of our simulator consists of NVidia TITAN V GPU based on Volta architecture, which has 80 SMs and 12 GB GPU Memory. The benchmarks used in our experiments were generated by increasing the number of kernels to fulfill the desired utilization of GPGPU. The number of thread blocks for each kernel ranges from 12 to 128. The execution time and the memory requirement of each thread block follow the uniform distribution ranging 5 to 20 msec and 1 KB to 1 MB, respectively. Under these GPGPU workloads, we conduct simulation experiments with the three scheduling policies, Round-Robin (RR), Breadth First Allocation (BFA), and Depth First Allocation (DFA). For performance metric, we use the Average Normalized Turnaround Time (ANTT) like previous studies [24,25]. ANTT measures the execution time of a task including the resource overhead incurred by the interference of other tasks and normalizes it to the basic execution time of the target task. For example, if the ANTT is 2, it takes twice as much time as compared to the case where the work is performed alone. Thus, a small ANTT value implies the better performance of the scheduling policy.

Figure 6 shows the ANTT of the three scheduling policies as the desired utilization is 10% and 80%, respectively. As shown in the figure, the performance gap is not wide when the desired utilization is 10%. However, as the desired utilization becomes 80%, the performance of Round-Robin is degraded significantly. Specifically, the ANTT of Round-Robin is 2.3 when the desired utilization is 80%. Note that the performance degradation of Round-Robin is over 70% compared to BFA or DFA. This implies that the current scheduling policy supported in GPGPU hardware is not efficient when the workload becomes heavy, and efficient scheduling is necessary.

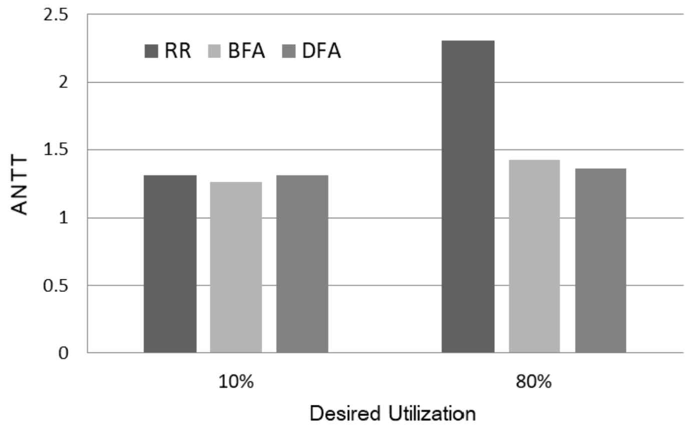

Figure 6. Performance comparison of the scheduling policies as the desired utilization is varied.

Figure 7 shows the actual utilization of SM as the scheduling policies and the desired utilization are varied. As shown in the figure, the actual utilization of the three policies is significantly lower than the desired utilization. This is due to the internal fragmentation of the workloads. When the workload is not heavy, the utilizations of the three policies exhibit similar results. However, as the workload becomes heavy, the actual utilization of Round-Robin is about 8% less than that of the other policies. Compared to the result in utilization, the gap in ANTT was very large; hence, it can be seen that increasing the utilization through efficient thread block scheduling is important to enhance the actual performance.

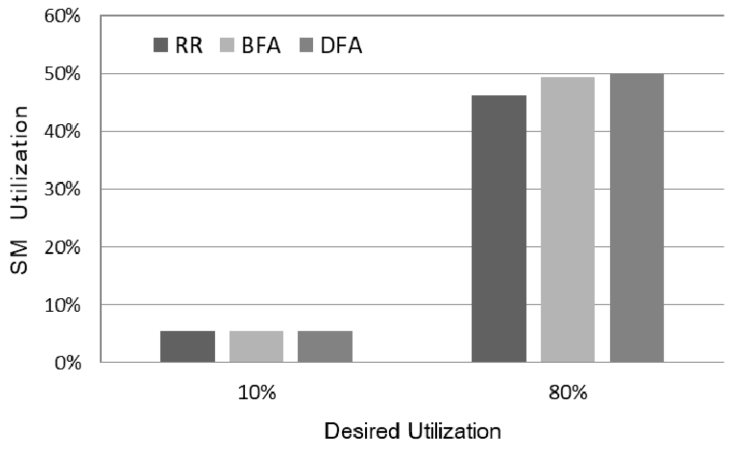

Figure 7. SM utilizations of the scheduling policies as the desired utilization is varied.

Figure 8 shows the performance of BFA and DFA relative to Round-Robin as the desired utilization is largely varied. Although we increase the desired utilization to 125%, which exceeds the total computing resources, it can be seen that the actual utilizations of BFA and DFA are only 76.1% and 76.4%, respectively, which are within acceptable ranges. Thus, it is meaningful to see the performance trend when the desired utilization is as large as 125%. However, as shown in Figure 8, ANTT increases steeply as the desired utilization becomes over 100%. Also, when comparing BFA and DFA, it can be seen that DFA performs even better than BFA when the load becomes extremely high.

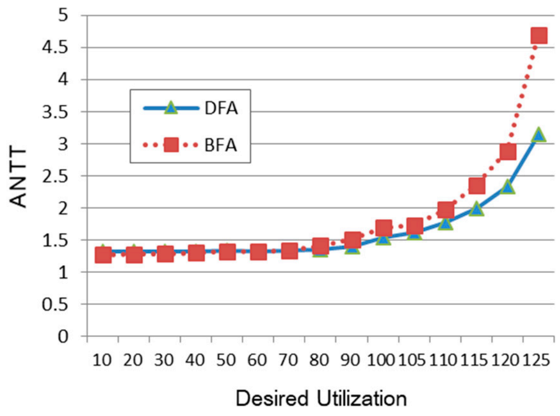

Figure 8. Performance comparison under workloads of uniform resource demand.

Figure 9 shows the performance of BFA and DFA similar to Figure 8, but the workload environment has extremely biased resource demand. That is, if there are two GPGPU tasks, one has the computing resource demand of 1 per SM, whereas the other has the maximum computing resource demand. In this case, as the desired utilization becomes high, it is difficult to allocate resources to the task with higher resource demand. Specifically, as the desired utilization becomes higher than 100%, the ANTT of BFA steeply increases, but that is not the case for DFA. Even though the desired utilization becomes 120%, the ANTT of DFA is still less than 4. Thus, we can summarize that DFA exhibits scalable performance, although the load of the task becomes high and the workload is skewed as it makes use of the maximum utilization of SM efficiently.

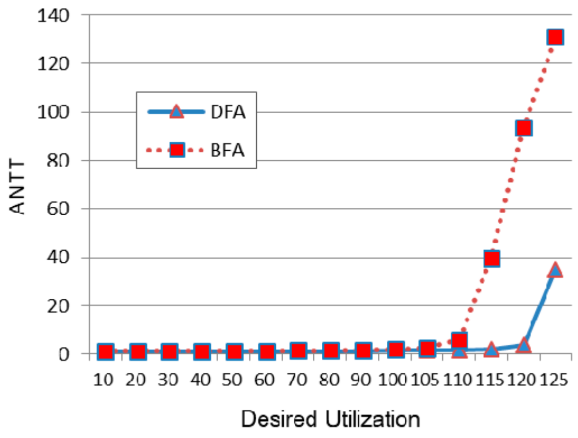

Figure 9. Performance comparison under workloads of extremely biased resource demand.

5. Conclusions

In this article, we presented a scheduler model for the thread block scheduling in GPGPU and developed an open-source GPGPU simulator based on the proposed model. We observed that the current Round-Robin hardware scheduler used in GPGPU is not efficient when the workload of GPGPU increases. In particular, the current hardware scheduler is simple and fast, and it shows reasonable performances when the load of GPGPU is not heavy [26]. However, when a large number of GPGPU tasks are requested simultaneously and the throughput and the execution time for multiple tasks are important, an efficient thread block scheduling is required to maximize the utilization of SM [27]. The proposed thread block scheduling model manages the available resources for each thread block by the unit of micro thread block (mTB), and evaluates the execution overhead based on the usage of the computing and memory resources. Unlike gpgpusim [28,29], which accurately emulates hardware in component basis, the proposed model simulates the characteristics of GPGPU with respect to the thread block scheduling. Experimental results using our simulator showed that BFA and DFA perform over 70% better than Round-Robin by making use of the resource usage in SM efficiently as the load of GPGPU increases.

In reality, the commercial GPGPU device does not open their interface for thread block scheduling, and thus it is not possible to implement our model in real GPGPU devices. Thus, developing simulators is an alternative way for evaluating the thread block schedulers. As our future study, we plan to implement an efficient thread block scheduler based on machine learning for considering the workload characteristics. For example, the scheduling performance can be improved even more by considering the heaviness of workloads and the homogeneity/heterogeneity of resource demands.

Author Contributions: K.C. implemented the architecture and algorithm and performed the experiments. H.B. designed the work and provided expertise. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the ICT R&D program of MSIP/IITP (2018-0-00549, Extremely scalable order preserving OS for manycore and non-volatile memory).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Nickolls, J.; Dally, W.J. The GPU Computing Era. *IEEE Micro* **2010**, *30*, 56–69. [CrossRef]
2. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M. Tensorflow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, Savannah, GA, USA, 2–4 November 2016; pp. 265–283.
3. Maitre, O.; Lachiche, N.; Clauss, P.; Baumes, L.; Corma, A.; Collet, P. Efficient parallel implementation of evolutionary algorithms on GPGPU cards. In *European Conference on Parallel Processing*; Springer: Berlin/Heidelberg, Germany, 2009. [CrossRef]
4. Garland, M. Parallel computing with CUDA. In Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS), IEEE Computer Society, Atlanta, GA, USA, 19–23 April 2010. [CrossRef]
5. CUDA Toolkit Documentation. Available online: https://docs.nvidia.com/cuda/ (accessed on 19 December 2020).
6. Lee, S.; Arunkumar, A.; Wu, C. CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. In Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, USA, 13–17 June 2015; Volume 43, pp. 515–527.
7. Ryoo, S.; Rodrigues, C.; Baghsorkhi, S.; Stone, S.; Kirk, D.; Hwu, W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, UT, USA, 20–23 February 2008.
8. Galvin, P.; Gagne, G.; Silberschatz, A. *Operating System Concepts*; John Wiley & Sons: Hoboken, NJ, USA, 2003.
9. Mostafa, S.; Rida, S.Z.; Hamad, H.S. Finding time quantum of round robin CPU scheduling algorithm in general computing systems using integer programming. *Int. J. Res. Rev. Appl. Sci.* **2010**, *5*, 64–71.
10. Shreedhar, M.; Varghese, G. Efficient fair queuing using deficit round-robin. In Proceedings of the ACM SIGCOMM Computer Communication Review, New York, NY, USA, 3 July 1996; Volume 4, pp. 375–385. [CrossRef]
11. Nieh, J.; Vaill, C.; Zhong, H. Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler. In Proceedings of the 2001 USENIX AnnualTechnical Conference, Boston, MA, USA, 25–30 June 2001. [CrossRef]
12. TBS Simulator. Available online: https://github.com/oslab-ewha/simtbs (accessed on 19 December 2020).

13. Lee, J.; Lakshminarayana, N.B.; Kim, H.; Vuduc, R. Many-thread aware prefetching mechanisms for GPGPU applications. In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, GA, USA, 4–8 December 2010.
14. Yunjoo, P.; Donghee, S.; Kyungwoon, C.; Hyokyung, B. Analyzing Fine-Grained Resource Utilization for Efficient GPU Workload Allocation. *J. Inst. Internet Broadcasting Commun.* **2019**, *19*, 111–116. [CrossRef]
15. Su, C.-L.; Chen, P.Y.; Lan, C.-C.; Huang, L.-S.; Wu, K.H. Overview and comparison of OpenCL and CUDA technology for GPGPU. In Proceedings of the 2012 IEEE Asia Pacific Conference on Circuits and Systems, Kaohsiung, Taiwan, 2–5 December 2012.
16. Jia, Z.; Maggioni, M.; Smith, J.; Scarpazza, D.P. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. *arXiv Preprint* **2019**, arXiv:1903.07486.
17. Wang, Q.; Xiaowen, C. GPGPU performance estimation with core and memory frequency scaling. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 2865–2881. [CrossRef]
18. Andersch, M. Analyzing GPGPU pipeline latency. In Proceedings of the International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), Fiuggi, Italy, 13–19 July 2014.
19. Orzechowski, P.; Boryczko, K. Effective biclustering on GPU-capabilities and constraints. *Prz Elektrotechniczn* **2015**, *1*, 133–136. [CrossRef]
20. López-Fernández, A.; Rodriguez-Baena, D.; Gomez-Vela, F.; Divina, F.; Garcia-Torres, M. A multi-GPU biclustering algorithm for binary datasets. *J. Parallel Distrib. Comput.* **2021**, *147*, 209–219. [CrossRef]
21. Collange, S.; Daumas, M.; Defour, D.; Parello, D. Barra: A parallel functional simulator for GPGPU. In Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Miami Beach, FL, USA, 17–19 August 2010.
22. Wang, Z.; Yang, J.; Melhem, R.; Childers, B.; Zhang, Y.; Guo, M. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, Spain, 12–16 March 2016.
23. Xu, Q.; Jeon, H.; Kim, K.; Ro, W.W.; Annavaram, M. Warped-slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. In Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Korea, 18–22 June 2016.
24. Park, J.J.K.; Park, Y.; Mahlke, S. Dynamic resource management for efficient utilization of multitasking GPUs. In Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Xi'an, China, 8–10 April 2017. [CrossRef]
25. Pai, S.; Thazhuthaveetil, M.J.; Govindarajan, R. Improving GPGPU concurrency with elastic kernels. In Proceedings of the ACM SIGARCH Computer Architecture News, Bangalore, India, 16 March 2013; Volume 41, pp. 407–418.
26. Badawi, A.A.; Veeravalli, B.; Lin, J.; Xiao, N.; Kazuaki, M.; Mi, A.K.M. Multi-GPU Design and Performance Evaluation of Homomorphic Encryption on GPU Clusters. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *32*, 379–391. [CrossRef]
27. Yeh, T.T.; Sabne, A.; Sakdhnagool, P.; Eigenmann, R.; Rogers, T.G. Pagoda: Fine-grained GPU resource virtualization for narrow tasks. In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, 4–8 February 2017; Volume 52, pp. 221–234. [CrossRef]
28. Aaron, A.; Fung, W.W.L.; Turner, A.E.; Aamodt, T.M. Visualizing complex dynamics in many-core accelerator architectures. In Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), White Plains, NY, USA, 28–30 March 2010. [CrossRef]
29. Hughes, C.; Green, R.; Voskuilen, G.R.; Zhang, M.; Rogers, T. GPGPU-Sim Overview. In *Technical Report, No. SAND2019-13453C*; Sandia National Lab: Albuquerque, NM, USA, 2019.