SSRG International Journal of Electronics and Communication Engineering
ISSN: 2348-8549/ https://doi.org/10.14445/23488549/IJECE-V11I5P111

Volume 11 Issue 5, 108-114, May 2024
© 2024 Seventh Sense Research Group®

Original Article

Impact of Different Thread Block Sizes with Synchronization and Shared Memory on the Performance of GPGPU

Sonal John[1], Saurabh Jain[2]

[1,2]Shri Vaishnav Institute of Computer Applications, Shri Vaishnav Vidyapeeth Vishwavidyalaya, Indore, Madhya Pradesh, India.

[1]Corresponding Author : sonaljohn@svvv.edu.in

Received: 15 March 2024 Revised: 18 April 2024 Accepted: 12 May 2024 Published: 31 May 2024

Abstract - *The Graphics Processing Units (GPUs) have many cores, and so give an improved execution level throughput. GPUs are intended for use in parallel computing. The size of a thread block plays a crucial role in determining the kernel's occupancy since thread-level parallelism is necessary to maximize overall performance. During the kernel launch, information on the number of threads per block and the number of blocks in a grid was provided. The variance in thread block sizes and the number of thread blocks within a grid greatly influence CUDA application performance. The impact of different thread block sizes with shared memory and synchronization on the total execution time of a few CUDA programs has been noted in this proposed work, along with a speed optimization. Additionally, implementing Shared Memory along with Synchronized Thread Blocks improves CUDA applications' overall performance in a GPGPU measurably.*

Keywords - *GPGPU, Synchronization, Shared Memory, Thread Block, Parallelism.*

1. Introduction

Multicore CPUs and many core GPUs have appeared and gradually established state-of-the-art high-performance computing. While CPUs consist of multiple cores and are utilized for high-performance parallel processing, GPUs contain hundreds of cores for dedicated tasks. Although the semiconductor technology used to make modern CPUs and GPUs is the same, GPU computational performance is increasing faster than CPU computational performance. Since each processing element has more transistors dedicated to control logic like branch prediction and out-of-order execution, CPUs are intended for high-performance, task-parallel workloads. In arithmetic, logics like floating-point require more transistors, and GPUs are more suitable for performing parallel tasks [1].

Advances in programming and economical architectural designs have made GPU's deployment more global. Created for graphics-related jobs in the early 2000s, GPUs were later enhanced to produce more efficiently, opening up new opportunities for developers. This led to the development of NVIDIA's Compute Unified Device Architecture (CUDA), a more flexible computing platform and programming language. The scientific research community soon adopted CUDA because it allowed high-level programming languages like C to be used to program NVIDIA GPUs. A GPU program can be written by programmers using the CUDA programming model as a kernel function that defines a 1D–3D array of thread blocks known as Cooperative Thread Arrays (CTAs), where each thread runs the identical code. Scheduled and performed on GPU compute units known as Streaming Multiprocessors (SM) are thread blocks [2, 3].

The host code used to launch kernels can manage thread-level parallelism through the execution configuration specifications. The number of thread blocks and the number of threads per block in the kernel launch are specified in the execution configurations. An important aspect, which a variety of reasons may constrain, is the maximum number of thread blocks that a multiprocessor can support for a given kernel. No matter how big the thread block is or how many resources are used, the number of thread blocks that can be used on a multiprocessor is limited. So, the GPU thread blocks play an important role in parallelizing an application and CUDA kernel launch. The objective of the proposed study is to:

- Observe the effect of different GPU Thread Block sizes on the CUDA applications [4, 5].
- Implement and analyze the efficiency of CUDA applications using the combination of shared memory with thread synchronization.

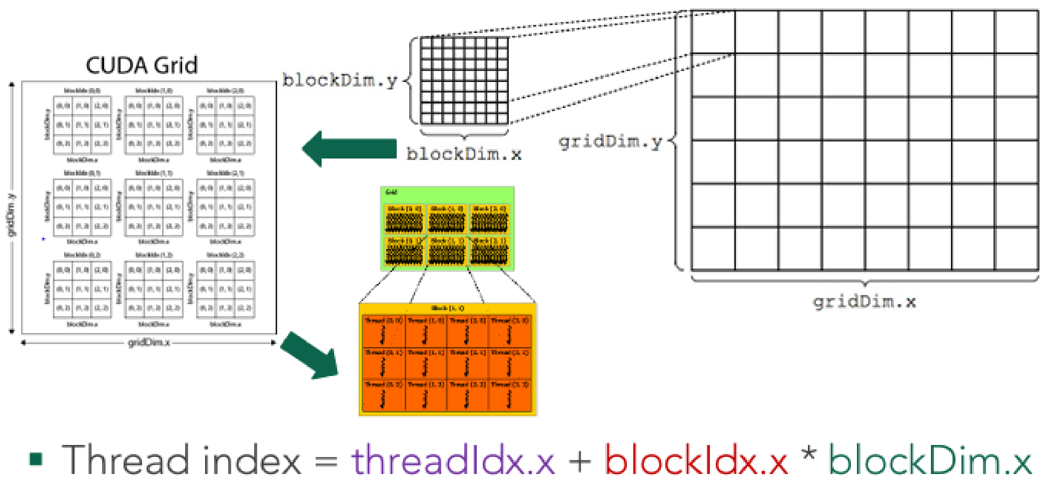

Fig. 1 Architecture of blocks, grids, and threads [6]

Accessibility to shared memory is significantly faster than that of global memory since it is an on chip memory. Shared memory, which is shared by every thread in a thread block, gives a way for thread cooperation. Every thread in the block gets access to the same amount of shared memory because it is allocated per thread block.

The data shared by a thread into the shared memory from global memory can be accessed by any other thread inside the same thread block. When thread synchronization is combined with the shared memory concept, it can be applied to several tasks, such as high-performance cooperative parallel algorithms, user-managed data caches, and enabling global memory coalescing in scenarios where it was not feasible [7].

2. Review of Literature

Improved floating point performance has been a major emphasis of GPU development. A massively parallel architecture known as CUDA, which NVIDIA introduced in 2006–2007, changed the world of GPU programming. Applications can be executed with greater efficiency because of the use of CUDA's multiple cores to handle data sets. Ghorpade et al. [8] address common assumptions regarding CUDA and its future aspects along with its architecture and also compare it with other parallel programming languages like OpenCL and DirectCompute.

The GPUs provide the synchronization in two different levels: warp level and thread block level. However, thread level synchronization is becoming necessary due to the increasing number of parallel applications running in a multithreading environment.

A thread level synchronization method termed lock stealing was proposed by Gao et al. [9] to prevent the threads in a warp from circular locking. The suggested method is utilized for reader-writer locking and mutual exclusion locking in order to enhance the overall throughput of the SIMT architectures.

Memory Access optimization is also a bigger challenge in parallel programming. In order to map the threads, Ohno et al. [10] suggested a memory access optimization approach in which first, the logical mapping is created. Then, it is further converted to physical mapping through the compiler in an optimized way. It is based on the static analysis of array index expressions and also improves the usage of physical resources.

To obtain Thread Level Parallelism (TLP), Li and Liang [11] presented an optimization system in this work that controls GPU concurrent kernel execution. The framework uses two essential methods. To modify the TLP for the kernel that is running concurrently, a TLP modulation method is created and also a cache bypassing method is designed to adaptively reduce the amount of thread blocks using the cache in order to reduce cache contention. The mentioned methods helped to improve the performance in the concurrent kernel execution environment.

Hong and Kim [12] implemented an analytical model that determines the memory operation cost, which is essential in determining how well parallel GPU applications perform. The estimation of a number of parallel memory requests, termed also as Memory Warp Parallelism (MWP) can be executed concurrently based on the number of threads that are executing concurrently and the memory bandwidth consumption.

As the appropriate thread block size to gain performance optimization in terms of GPU occupancy is challenging, and so Connors and Qasem [13] proposed a machine-learning technique to identify the profitable block sizes automatically. Furthermore, the study also demonstrated how the combination of performance counters and machine learning algorithms can reveal the fundamental causes of performance variance between various configurations.

To facilitate the dynamic launching of lightweight thread blocks, Wang and Rubin [14] proposed a new technique termed DTBL (Dynamic Thread Block Launch), which supports the GPU execution model. This approach allows dynamically arriving parallel tasks to be executed by multiple thread blocks instead of kernels. The DTBL execution paradigm, device-runtime support, and microarchitecture enhancements for monitoring and executing dynamically created thread blocks.

Aguilera et al. [15] proposed various approaches to define the term fairness for GPGPU spatial multitasking by measuring the performance of each application. The study suggested a number of resource distribution schemes and also provided a run-time algorithm that predicts and modifies the SM allocation to satisfy the required fair share specifications. In this study, the problem of fair computing resource distribution among apps running on spatially multitasked GPUs is investigated.

Hijma et al. [16] analyzed the different optimization techniques and also gave an overview of these techniques to improve the overall performance of the system. Also the impact of the evolution of architectures on the performance of different applications is discussed in the study.

Table 1. Analysis of different studies

References	Finding	Limitations
[8]	A comparison of the CUDA programming model with other languages like OpenCL and DirectCompute has been done. Also the future aspect of CUDA in terms of performance improvement of GPGPU applications is done.	Authors have focused on only NVidia's architecture, but Nvidia still has many difficulties to meet to make CUDA stick, since while technologically it is undeniably a success.
[9]	The authors have provided the synchronization in two different levels which are Thread level and wrap level. Authors have suggested to utilize for reader-writer locking and mutual exclusion locking in order to enhance the overall throughput of the SIMT architectures, which is one of the major parameters.	The author has focused on developing techniques to avoid live-locks, but a try-lock scheme is not used, and that is a major limitation.
[10]	A memory access optimization technique is suggested in which an optimized mapping of logical memory to physical memory is done through the compiler.	The study is done for a few small benchmarks only and so for more fair estimation, more improvisation in the current scheme is required.
[11]	The major finding provided by the author is to improve the performance in the concurrent kernel execution environment using two TLP methods.	The authors do not optimize the Thread Level Parallelism (TLP) and model the resource contention for the concurrently executing kernels.
[12]	Implementation of a model for estimating the overall time taken by parallel memory operations in parallel which is known as Memory Warp Parallelism (MWP).	The work is focused only on memory intensive GPU applications.
[13]	The authors proposed a machine learning technique to identify the profitable block sizes automatically. The author also finds that a combination of performance counter and machine learning algorithms is the fundamental cause.	ML models have specific algorithms which support hardware support, so that some issues may occur in future.
[15]	The authors have presented the fairness of computer resource allocations for multitasking GPUs.	The main limitation of the proposed study is to demonstrate how to divide the computing resources among the concurrent executing applications.

3. Problem Description

The performance of a GPU's Streaming Multiprocessors (SM) depends on the Thread Level Parallelism (TLP), i.e. the number of threads present in a thread block and the number of thread blocks in an SM. However, the amount of shared memory and registers that each thread block uses determines how many thread blocks can be launched on an SM. Also, the size of a thread block is an important factor in the kernel launch and the performance improvement, so when the size of a thread block is changed, there is a change in the overall performance of an application executing on a GPU.

However, a significant amount of increase in the size of a thread block should be made because when the size is too short, more thread blocks will be there, and so synchronization with a large number of thread blocks will be difficult in terms of processing and time. If the size is too large, the parallelization will be reduced. The shared memory is faster as it is an on chip memory and is allocated per thread block. So the total amount of elapsed time during the kernel execution has a significant improvement while using the shared memory as compared to global memory. Also, thread synchronization is an important aspect when the increasing number of threads is used. So, in the present study, the combination of thread synchronization using shared memory has been implemented [7, 12].

3.1. Effect of Thread Block Size

At the time of a *kernel launch (_kernel<<<numBlocks, threadsPerBlock>>>(input, output))*, the Thread Level Parallelism can be achieved by varying the size of a thread block and seeing the effect of different thread block sizes over the kernel execution time, some combinations of block sizes (8 x 8, 16 x 16 and 32 x 32) have been taken. This is one of the key factors for kernel occupancy and GPU utilization [6].

3.2. Effect of Shared Memory

The effect of using shared memory as compared to global memory is to get optimized results as it is allocated to every thread block separately, and all the threads of a thread block share the same address space so that thread cooperation can be seen. In CUDA programming using *_shared_* identifier, a speed up in the overall performance of the application can be found [7].

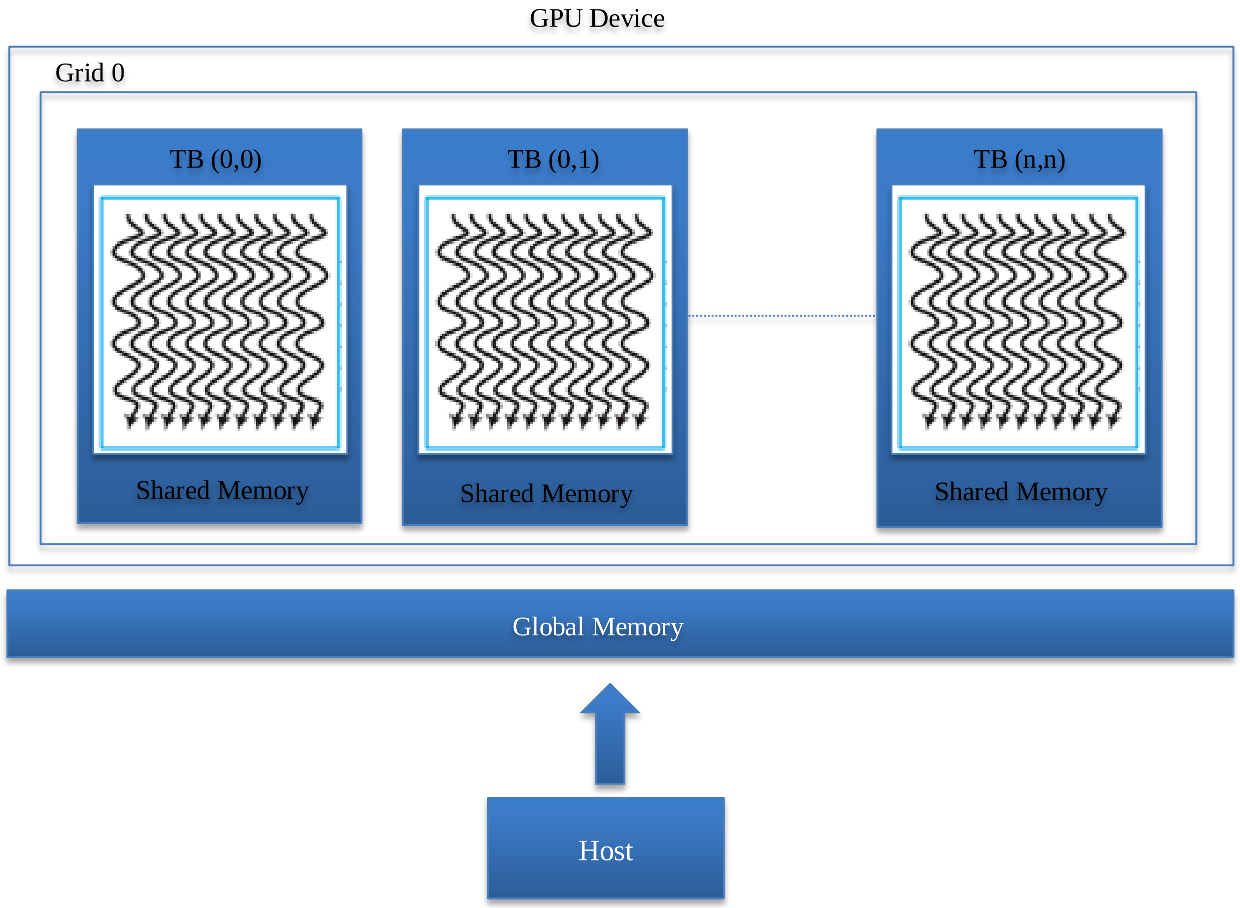

Fig. 2 Thread block architecture with shared memory (self-made)

3.3. Effect of Thread Synchronization

In a CUDA application, the synchronization of threads can be obtained through the *_syncthreads()* method. It is a barrier that makes sure that no thread in the block can execute until every thread has passed through the barrier. Additionally, it is ensured that all writes to memory done by threads in the block before and after the barrier will be available to these threads [12].

4. Proposed Work

In this paper, the performance of GPGPU has been analyzed by using different thread block sizes in various CUDA applications and found the most significant thread block size that can be used depending on the type and data size of the application. Also, the combination of thread synchronization using shared memory has been implemented to get more optimized results.

5. Experimental Setup

A device named GeForce GTX 1050 Ti with a computing capacity of 6.1, a maximum of 2048 threads per SM and a maximum of 1024 threads per block is used for the experiment. It has the Pascal architecture. CUDA 10.1 is used for the hardware programming.

To analyze the impact of using shared memory on the performance of the system in terms of total elapsed time of kernel execution, two different matrix based applications, Matrix Multiplication (MM) and Matrix Transpose (MT), with different thread block sizes, have been taken into consideration. For the experiment, three different sizes of thread blocks (8 x 8 = 64), (16 x 16 = 256) and (32 x 32 = 1024) are taken. On the other hand one more experiment has been done on the same applications in CUDA using Global and Shared memories.

The impact of varying sizes of thread blocks can be seen in the following graphs:

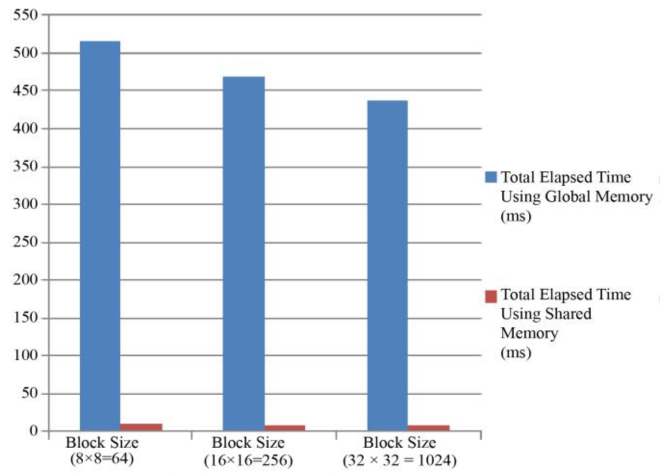

Fig. 3 Matrix size 1024 x 1024 for MM

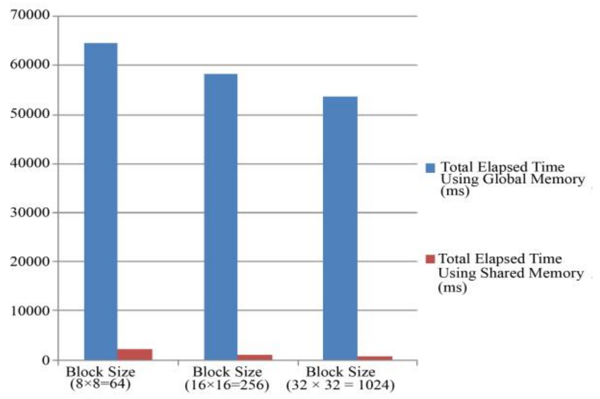

Fig. 4 Matrix size 5120 x 5120 for MM

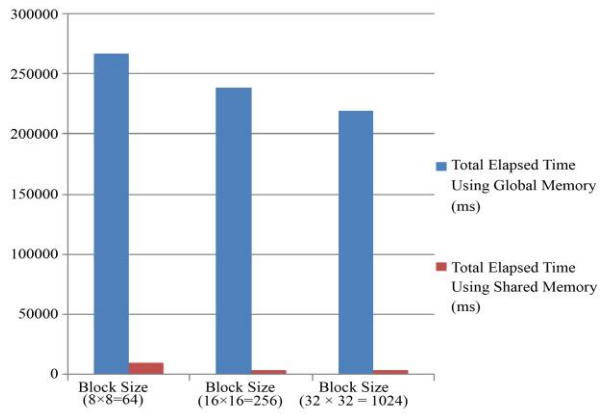

Fig. 5 Matrix size 8192 x 8192 for MM

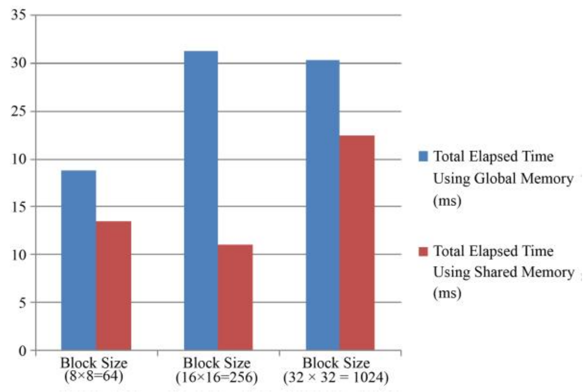

Fig. 6 Matrix size 1024 x 1024 for MT

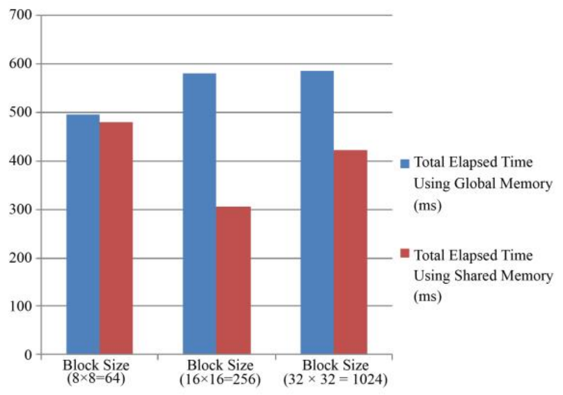

Fig. 7 Matrix size 5120 x 5120 for MT

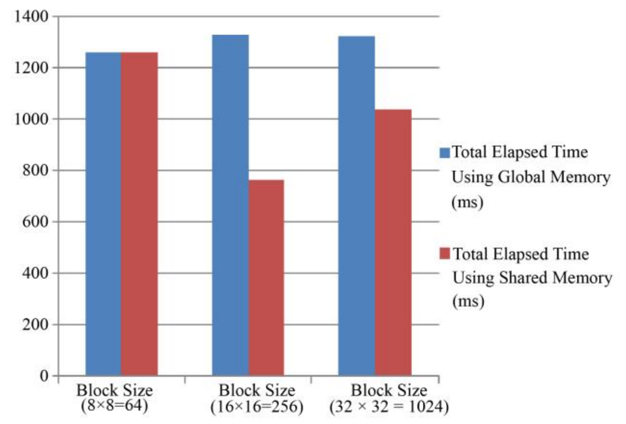

Fig. 8 Matrix size 8192 x 8192 for MT

6. Results and Discussion

The graphs obtained from the three datasets on both applications depict some interesting and significant observations. For the matrix multiplication application, all three graphs (4.1, 4.2 and 4.3) show the same pattern for both the cases of global and shared memory. While considering the block size of 8 x 8, the total elapsed time is higher.

When block size 16 x 16 is considered for all three datasets, it has been observed that the total elapsed time has drastically decreased, particularly with shared memory. However, with the block size of 32 x 32, the total elapsed time is showing some slight decline but this can also be seen while executing the application repeatedly. Therefore, block size 32 x 32 does not have any significant effect on the total elapsed time.

With the application of matrix transpose, almost the same pattern can be observed through the graphs, especially in the case of shared memory. Interestingly, in all three datasets, it is observed that the total elapsed time using shared memory has increased with the increase in block size from 8 x 8 to 16 x 16, and almost the same effect can be observed with block size 32 x 32. Therefore, there is no sense in increasing the block sizes. When the shared memory is used, it can be seen that with all three datasets, the block size of 16 x 16 shows the optimized results.

With the help of different matrix sizes from 1024 X 1024 to 8192 X 8192, the performance of the CUDA applications in terms of global and shared memory is compared and also observed the impact of change in the size of a thread block.

7. Conclusion

This research paper has been undertaken to analyze the impact of using shared and global memory and also variations in the size of thread blocks. Here, two applications are taken into consideration with Thread Level Parallelism, and their effect on the overall kernel execution time has been observed on a GPU device. It can be significantly seen that the use of shared memory along with global memory provides better results. Both the applications also show the same impact that moderate block size gives optimized results in terms of the total elapsed time of kernel execution. Further, this paper can be useful for researchers and even for operating system designers so that when the thread block size is moderate, more elements can be processed in parallel. If a large number of threads will be accommodated in a single thread block, the parallelization will be reduced.

References

[1] Liang Hu, Xilong Che, and Si-Qing Zheng, “A Closer Look at GPGPU”, *ACM Computing Surveys*, vol. 48, no. 4, pp. 1-20, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[2] Richard Vuduc, and Jee Choi, *A Brief History and Introduction to GPGPU*, Modern Accelerator Technologies for Geographic Information Science, Springer, New York, pp. 9-23, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[3] John Nickolls, and William J. Dally, “The GPU Computing Era”, *IEEE Micro*, vol. 30, no. 2, pp. 56-69, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[4] Massimiliano Fatica, and Gregory Ruetsch, *CUDA Fortran for Scientists and Engineers*, Best Practices for Efficient CUDA Fortran Programming, pp. 43–114, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[5] Vishwesh Jathala, “*Hardware and Software Optimizations for GPU Resource Management*”, Ph.D Thesis, Kanpur, India, 2018. [Google Scholar] [Publisher Link]

[6] H. Harmanani, “Parallel Programming for Multi-Core and Cluster Systems CUDA Thread Scheduling, 2018. [Online]. Available: https://harmanani.github.io/classes/csc447/Notes/Lecture15.pdf

[7] Mark Harris, “Using Shared Memory in CUDA C/C++”, 2013. [Online]. Available: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[8] Jayshree Ghorpade et al., “GPGPU Processing in CUDA Architecture”, *Advanced Computing: An International Journal*, vol. 3, no. 1, pp. 105-120, 2012. [CrossRef] [Google Scholar] [Publisher Link]

[9] Lan Gao et al., “Thread-Level Locking for SIMT Architectures”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 5, pp. 1121-1136, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[10] Kazuhiko Ohno et al., “Automatic Optimization of Thread Mapping for a GPGPU Programming Framework”, *2014 Second International Symposium on Computing and Networking*, Shizuoka, Japan, pp. 198-204, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[11] Xiuhong Li, and Yun Liang, “Efficient Kernel Management on GPUs”, *IEEE 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, pp. 85-90, 2016. [Google Scholar] [Publisher Link]

[12] Sunpyo Hong, and Hyesoon Kim, “An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness”, *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pp. 152–163, 2009. [CrossRef] [Google Scholar] [Publisher Link]

[13] Tiffany A. Connors, and Apan Qasem, “Automatically Selecting Profitable Thread Block Sizes for Accelerated Kernels”, *IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Bangkok, Thailand, pp. 442-449, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[14] Jin Wang et al., “Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs”, *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp 528-540, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[15] Paula Aguilera, ,Katherine Morrow, and Nam Sung Kim, “Fair Share: Allocation of GPU Resources for both Performance and Fairness”, *IEEE 32nd International Conference on Computer Design (ICCD)*, Seoul, Korea (South), pp. 440-447, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[16] Pieter Hijma et al., “Optimization Techniques for GPU Programming”, *ACM Computing Survey*, vol. 55, no. 11, pp. 1-81, 2023. [CrossRef] [Google Scholar] [Publisher Link]