Introduction to CUDA GPU Programming

Koushik Sivarama Krishnan
V01031395
koushik0901@uvic.ca
University of Victoria

ABSTRACT

This report is a compilation of the understanding and application of the fundamentals of CUDA GPU using C++. Firstly, a simple matrix multiplication program is written in C++ and optimized to incorporate parallel computing in CUDA. This program was then changed to use more threads for each corresponding output element of the resultant matrix; this caused race conditions, which were solved using atomic operations, parallel reduction, and warp-level primitives. The effectiveness of these different approaches to eliminate race conditions was then measured and stated. This report also briefly compares tensor and CUDA cores with a view to understanding their suitability in parallel computation. The study's results, therefore, affirm the effectiveness of CUDA programming in enhancing computational operations, especially in machine learning.

ACM Reference Format:
Koushik Sivarama Krishnan. 2024. Introduction to CUDA GPU Programming. In *Proceedings of University of Victoria (Koushik Sivarama Krishnan).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

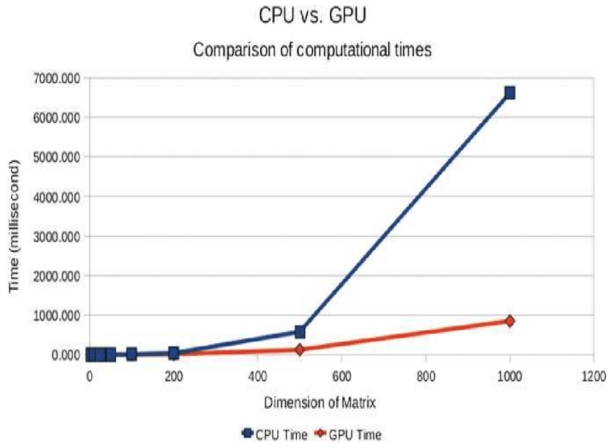

Figure 1: Comparision of GPU vs CPU for matrix multiplication task [9]

1 INTRODUCTION

Over the last few years, GPUs have drastically upgraded their ability to perform parallel computing efficiently. GPUs are fundamentally different from CPUs in terms of their use and architectures. CPUs are optimized for general-purpose computing and can perform tasks exceptionally well in a sequence by having low-latency access to memory and efficient context switching. A typical CPU consists of few powerful cores that are optimized for processing tasks sequentially and is best suited for tasks that involve complex decision-making processes.

GPUs are intended to perform very well in parallel processing. They include thousands of less complex cores that can handle many threads at the same time. This design makes GPUs very suitable for tasks that can be divided into smaller and more independent tasks, such as matrix multiplication, image processing, rendering, and deep learning.

CUDA is a parallel computing application programming interface model invented by NVIDIA. It is the foundation of NVIDIA's GPUs and enables developers to tap into the massive computing capabilities of the GPUs. CUDA adds to the existing programming languages like C and C++ to be able to write CUDA programs that can be executed on the GPUs, thus enhancing the performance of computations involved in rendering and deep learning in several ways. [3]

A primary idea in CUDA programming is the Single Instruction Multiple Thread (SIMT) model of parallelism that NVIDIA made famous. This SIMT model enables one instruction to coordinate many threads simultaneously to accomplish a task. It eases parallel programming development since it hides the details of handling threads and synchronization. While utilizing parallel processing, the SIMT model is good; it can lead to race conditions where two or more threads try to read, write, or even modify the same memory address. This would lead to wrong results when executing parallelism of instructions in the computer.

Therefore, managing race conditions is central to any parallel programming application to guarantee the correctness of the output. This race condition can be managed using a number of measures, including atomic operations, parallel reductions, and warp-level primitives.

The last optimization technique that can help make the matrix multiplication process faster is using tensor cores on the GPU. Tensor cores are particular processors developed to speed up the matrix computation by doing several computations in a single clock cycle and high throughput of matrix-vector multiply. If we add tensor cores instead of CUDA cores in the kernel, the instruction count is significantly lower, and the program performs better.

The evaluation section of the report describes the outcomes of the performance enhancement experiments performed on the NVIDIA GPUs with the help of Google Colab. In this report, we will look at the uses of CUDA programming by beginning with the simplest

Koushik Sivarama Krishnan, July 05 2024, Victoria, BC

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

program, the "hello world" program to do matrix multiplication. We then introduce race conditions and apply the above mentioned techniques to solve them. By these milestones, the report provides an introduction and shows how CUDA programming can be used. It overviews the comparison between GPUs and CPUs to introduce the specificity of GPUs in parallel computing.

This report analyzes the above results to show that CUDA programming enhances computational procedures by comparing various practices in handling race conditions and their efficiency. Substantial gains in numerous computations can be obtained when developers know and apply the peculiarities of the GPU and the SIMT model.

This report shows how CUDA can perform parallel processing by creating a fundamental matrix multiplication problem and improving it through shared memory, register tiling, and tensor cores. The study also demonstrates that CUDA helps enhance computational operations, especially in deep learning, which includes many matrix operations.

2 BACKGROUND

Matrix multiplication is essential in deep learning and natural language processing. To better understand GPUs and how to utilize them, we wrote elementary CUDA code for matrix multiplication. This first solution serves as a starting point for comparing other optimization methods.

Thus, matrix multiplication can be parallelized efficiently by using CUDA because it provides data parallelism. A standard matrix multiplication operation in C++ involves multiplying two matrices: A and B. The result is stored in a third matrix, C, where each element in a matrix is computed as a dot product of the i-th row in matrix A and the jth row in matrix B. This computation is independent for each output element, which is ideal for parallel computing using GPUs.

In this simple example of matrix multiplication, each thread will calculate one element of the resulting matrix. This ensures the computation is fractional across the number of available cores in the GPU. All the data for matrices A, B, and C are stored in the global memory. All the threads read the data, do the computation, and store the data in the global memory. The CUDA kernel for this approach is launched with a grid and block configuration of NxM, which is the same as the dimensions of the resultant matrix C, NxM.

While this approach is about 200 times faster than the CPU-based approach, it does not fully use the GPU computational resources. The run time of this current approach can be further reduced by using more than one thread to calculate each element of the matrix for the output. But this would then cause race-condition problems. The following sub-sections elaborate on various approaches that can be used to synchronize multiple threads to compute the same output element, which can lead to race conditions.

3 METHODS

3.1 Atomic Operations

Atomic operations are one of the simple ways of dealing with race conditions in CUDA. They are intended to make the read, modify, and write operations on a shared variable occur atomically, i.e.. , occur without any intervention by other threads, thereby synchronizing the access to the memory location. In this approach, we will use the atomic function obtained from the CUDA library to increment a variable that multiple threads access safely. When it comes to matrix multiplication, the atomicAdd function allows for the correct addition of each contribution of threads without losing any updates resulting from concurrent access. [6] [1]

3.2 Parallel Reduction

The main idea behind parallel reduction is to store the intermediate results in the shared memory and synchronize threads within each block to perform reduction operation effectively.

Parallel reduction works by loading each thread's data into the shared memory. Then, threads within each block cooperate to perform a reduction operation, in our case, an addition operation on this shared data stored in the memory. This is repeated where the number of threads working is halved every time until only one thread is needed to hold that block's final reduced (summed) value. Finally, the results from each block are further reduced to obtain the globally reduced final output. [2]

By aggregating values in shared memory, parallel reduction reduces the number of atomic operations required. Moreover, the parallel reduction is more highly scalable than atomic operations for large volumes of data by utilizing faster-shared memory for intermediate computation and minimizing global memory access. Parallel reduction adds complexity to the implementation compared to atomic operations, as it requires careful management of shared memory and synchronization of threads within each block.

3.3 Warp-level primitives

Another technique that can eliminate the race condition when multiple threads work on one output element is warp-level primitives in CUDA. A warp is a set of 32 threads that can cooperate with other threads belonging to the same warp to perform operations in the best manner possible. It enables the threads within each warp to share information and data by using registers rather than shared memory or even synchronization. This technique involves shuffling of data, voting and reduction operations within the warps. In general, each thread in a warp has a space in the register which are private to the thread in the warp. The warp-level primitives allow all the threads in a warp to read the registers of other threads belonging to the same warp. Thus, this technique eliminates overhead associated with the shared memory access. Some common warp-level primitive operations are _shfl_down_sync, _shfl_up_sync, _shfl_xor_sync and so on. In the matrix multiplication task, to reduce the partial sum computed within a group, each thread in a group uses the `__shfl_down_sync()` function to perform warp-wide reduction within each warp. This function adds and shifts the value down the warp and sums partial results across threads in the warp.

3.4 Tensor cores

Tensor cores are a major leap forward in parallel computer processing because they provide huge performance increases for matrix math, which are vital for any AI application. Tensor cores are the dedicated computing cores in NVIDIA GPUs which are optimized for matrix operations involved in deep learning operations. They

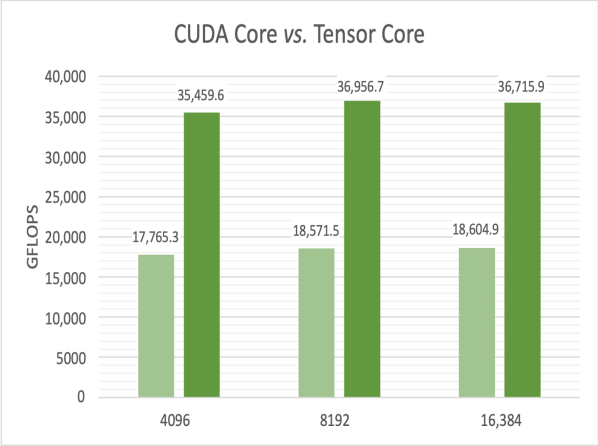

Figure 2: Comparision of CUDA and Tensor cores for matrix multiplication task [4]

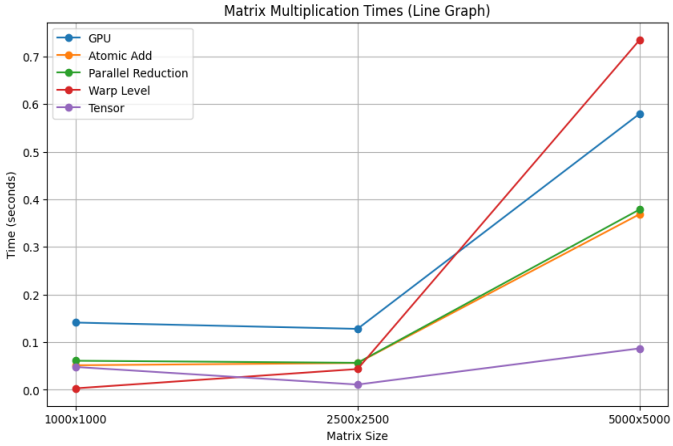

Figure 3: Comparision of different approaches to improve speed for matrix multiplication task

are much faster and more efficient compared to the previous CUDA cores for operations such as matrix multiplication and convolution operations used in deep learning. They incorporate mixed precision methods to handle matrices' multiplication operations in a single clock cycle. It runs matrix operations in both FP16 and FP32 mixed precision, allowing it to optimize speed and accuracy. At the same time, CUDA cores, as a rule, work in full precision of FP64 or FP32 to perform the same matrix operations and, therefore, can be slower when working with large matrices. We can utilize tensor cores with the help of CuBLAS library [7] in C++ as it is a library developed by NVIDIA for performing dense linear algebra on tensor cores. This library has high-level abstraction functions for matrix multiplication called cublasSgemm(), which will perform matrix multiplication on tensor cores. It internally determines the number of threads and blocks for effectively doing the matrix multiplication task.[8]

4 EVALUATION

This section details about the type of hardware used, how the matrices were initialized, and compares the performance of different techniques discussed above in the methods section. All the experiments were run on NVIDIA T4 GPU available on Google Colab [5]. The NVIDIA T4 GPU setup on Google Colab has a CUDA version 12.2 with 15360 MiB memory, 2560 CUDA cores, and 320 tensor cores. Each of the approaches discussed above have been timed across performing matrix multiplication operations on 1000x1000, 2500x2500 and 5000x5000 matrices. All these matrices are randomly initialized by setting a manual seed of 42 for reproducibility.

The standard CUDA implementation increases linearly as the matrix size increases. The atomic add shows a similar linear increase as the matrix size grows but this approach is significantly faster than the standard GPU implementation. Parallel Reduction method performs very similar to the Atomic add approach and is better than the standard GPU implementation. Both these approaches are faster because they are setup in a way that 16 threads contribute to computing one output element of the resultant matrix compared to the standard 1 thread per output element in the regular CUDA implementation. The Warp-level reduction operation performs better than all the other three approaches discussed above for 1000x1000 and 2000x2000 matrices but performs worse than the standard GPU implementation for 5000x5000 matrices. This could be due to increased synchronization overhead. Lastly, as expected, utilizing tensor cores for performing matrix multiplication performs significantly faster across all the matrix sizes compared to other approaches. This is expected as the tensor cores are specifically designed to optimize and efficiently perform matrix multiplication.

5 CONCLUSION

This report summarizes my learnings and findings of different approaches to efficiently perform matrix multiplication by utilizing CUDA and Tensor cores. We discuss different approaches to handling race conditions such as use Atomic operations, Parallel Reduction, and Warp-level Reduction. We implement all these techniques across 1000x1000, 2500x2500 and 5000x5000 matrices and by using 16 threads to operate on each output element in the resultant matrix. As, expected, implementation of matrix multiplication on tensor cores using the CuBLAS library performs significantly better than all other approaches. Followed by Atomic operations and Parallel Reduction approaches. Warp level primitive approach falls third since it takes longer to perform 5000x5000 matrix multiplication. The standard CUDA approach falls last since only one thread work on each output element. This module has been a significant learning experience of how to utilize CUDA and Tensor cores to perform parallel computing efficiently.

REFERENCES

[1] Michael Chow. 2020. GPU Programming Lecture Slides. https://www.cs.ucr.edu/~mchow009/teaching/cs147/winter20/slides/11-Histogram.pdf. Accessed: 2024-07-05.
[2] GPU Primitives Course. 2021. Home Page. https://gpu-primitives-course.github.io/. Accessed: 2024-07-05.
[3] Mark Harris. 2017. An Even Easier Introduction to CUDA. https://developer.nvidia.com/blog/even-easier-introduction-cuda/. Accessed: 2024-07-05.
[4] Xuanteng Huang, Xianwei Zhang, Panfei Yang, and Nong Xiao. 2023. Benchmarking GPU Tensor Cores on General Matrix Multiplication Kernels through CUTLASS. *Applied Sciences* 13, 24 (2023). https://doi.org/10.3390/app132413022

[5] Andrei Nechaev. 2021. nvcc4jupyter GitHub Repository. https://github.com/andreinechaev/nvcc4jupyter. Accessed: 2024-07-05.
[6] NVIDIA. 2014. GPU Pro Tip: Fast Histograms Using Shared Atomics on Maxwell. https://developer.nvidia.com/blog/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/. Accessed: 2024-07-05.
[7] NVIDIA. 2022. cuBLAS Library Documentation. https://docs.nvidia.com/cuda/cublas/. Accessed: 2024-07-05.
[8] Paperspace. 2021. Understanding Tensor Cores. https://blog.paperspace.com/understanding-tensor-cores/. Accessed: 2024-07-05.
[9] Snezhana Pleshkova and Al Bekiarski. 2018. *Development of Fast Parallel Algorithms Based on Visual and Audio Information in Motion Control Systems of Mobile Robots*. 105–138. https://doi.org/10.1007/978-3-319-67994-5_5