CS259-D: Accelerating Machine Learning Models with CUDA

Aditya Patel
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
aditya.s.patel@sjsu.edu

ABSTRACT
The goal of this paper is to explore ways to accelerate training machine learning models using parallel processing. It begins with an introduction to CUDA and the processes involved in training a neural network. The paper then focuses on the experiment setup, including the different configurations, workloads and programming optimizations implemented to accelerate the computations. These implementations are then benchmarked based on the running time to complete training. Then the paper analyzes, compares and draws insights on the results and trends observed in the performances under various conditions. The paper concludes with a guideline for choosing optimal configurations, and coding practices based on the task.

1. INTRODUCTION
The rapid growth in machine learning and deep neural networks [1] in recent years has led to unparalleled rise in computational demands. As datasets and model parameters continue to grow in size, traditional CPU-based computation struggles [2] to meet the performance requirements of large-scale model training and inference. To address these challenges, it has become essential to use parallel processing and computing technologies to accelerate machine learning workloads. NVIDIA's Compute Unified Device Architecture (CUDA) [3][4] is a leading platform to use the massive parallel processing power of Graphics Processing Units(GPUs). CUDA allows developers to exploit thousands of GPU threads to perform matrix operations and numerical computations by writing highly parallelized programs. Since neural networks training operations mainly comprise of matrix multiplications [5] and gradient updates, it aligns greatly with GPU's strengths which can be distributed efficiently across multiple cores. The training times of neural networks can be reduced greatly by leveraging CUDA, which allows fast computation, faster experimentation and improved model performance.

CUDA allows developers to explicitly control how these computations are distributed across GPU cores using concepts like threads, blocks and grids [6][7]. These configurations determine how data is partitioned and processed in parallel, influencing factors like occupancy, memory access efficiency, and overall execution time. However, determining optimal block sizes and grid sizes is not a trivial task. This is because the performance depends on kernel design, interaction between hardware resources, GPU block scheduling, and the model architecture. This paper investigates how different CUDA block and grid configurations affect the performance of training a neural networks. The study sets up controlled experiments with varying workloads and optimization strategies, measuring executions times to evaluate computational efficiency. The paper benchmarks these configs, and then analyzes how GPU resource utilizations and thread-level parallelism affects the overall runtime performance, The analysis highlights trends, tradeoffs, and best practices for optimizing CUDA based implementations of neural networks.

The paper will provide insights and guidelines for selecting the best CUDA configurations to help developers achieve maximum acceleration in machine learning computations.

2. BACKGROUND
2.1 CUDA
CUDA is a parallel computing platform and programming model created by NVIDIA that allows developers to parallelize operations and use NVIDIA GPUs for general-purpose processing [8][9]. CUDA, in itself, is a platform and is not limited to any one exclusive programming language, i.e., it can be integrated with multiple languages, including but not limited to CUDA C [8], CUDA Fortran, CUDA C++, etc. This project was implemented in CUDA C++, which will be synonymous with CUDA throughout the paper.

2.1.1 Kernels
Kernels are functions that are executed by multiple threads on the GPU [6]. A CUDA program typically has host code - that runs on the CPU, controls data, launches kernels, manages memory - and device code (kernels) that run on the GPU and perform computations in parallel. Kernels are marked by a special keyword __global__, which tells the compiler that the function will run on GPU.

2.1.2 Threads
Threads are the smallest unit of execution in CUDA. Each thread runs a single instance of a kernel function, and performs computations on a specific portion of data. Launching a kernel creates many threads that execute the same code simultaneously. This helps achieve massive data parallelism. Any number of threads can be run simultaneously to execute a kernel. Each thread has a unique ID that helps it determine which part of data to process. It is derived from threadIdx, blockIdx, and blockDim.

2.1.3 Blocks

A block is a group of threads that can cooperate with each other. Each block can contain up to 1024 threads (depending on the GPU architecture). Threads in the same block can communicate using shared memory and synchronize their execution using functions like __synchthreads(). Threads in different blocks, however, cannot directly communicate or synchronize with each other. Synchronization between blocks is limited but can be achieved using barriers or atomic operations. Blocks are the fundamental scheduling units in CUDA, and each block runs independently on one of the GPU's Streaming Multiprocessors (SMs) as shown in **Figure 1**.

2.1.4 Grids

Finally, blocks are organized into a grid. Grids are collection of blocks that are launched for a kernel. All the blocks in a grid are executed in parallel, however, only one grid can be executed at a time in a single stream. That is, grids in the same stream are executed in sequential manner. **Figure 1** demonstrates the relationship between threads, blocks and grids with respect to a kernel.

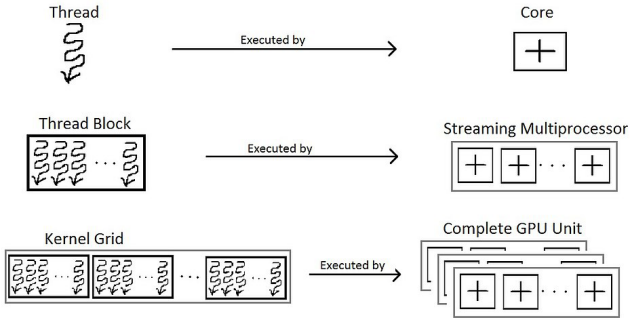

Figure 1. Relationship between threads, blocks and grids.

2.1.5 Memory Hierarchy

CUDA provides different memory types with varying access speeds and scopes, including global memory, shared memory, constant memory, texture memory, and registers [7][9]. Efficient memory management is the cornerstone of well-optimized CUDA programs. Unified memory in CUDA refers to a memory management system that allows for easily sharing memory between CPU and GPU by automatically handling data movement between processors [4]. This project utilizes Unified memory for a simpler, less complex implementation.

2.1.6 Memory Management

Proper memory management is essential to prevent memory leaks and ensure the safety of the program. In CUDA, cudaMallocManaged and cudaFree functions correspond to C++'s new and delete functionalities, respectively. Each pass maintains a set of intermediate values necessary to conduct automatic differentiation, which needs to be freed once an output has been returned.

2.2 Neural Networks

For this implementation of neural networks, the main building block is the Module class, which contains three main methods: forward, backward and update [10]. Every layer inherits this class and overrides these methods as necessary. The forward method transforms the input data by applying some operations on it. The backward method receives the upstream gradients which are derivative of the loss with respect to output of the layer, and the update method updates the parameters of the modules.

2.2.1 Linear layer

This module is a fully-connected network that employs the linear layers extensively. A linear layer performs matrix-multiplication on the input by a learnable matrix, called weights, and elementwise adds a vector called bias, to the result. One kernel might perform matrix multiplication and another elementwise-addition but they can be fused together for efficiency.

2.2.2 ReLU

Since multiple consecutive linear layers degenerate into a single linear layer, it is necessary to separate them using non-linearities to be useful [10]. The Rectified Linear Unit (ReLU) is a popular non-linear function that behaves like identity function for positive inputs and evaluates to 0 otherwise. This simplicity of ReLU means that it is very easy to calculate and has a straight-forward derivate: 1 for positive inputs and 0 otherwise.

2.2.3 Sequential

This module is used to automatically manage consecutive layers. A recurring pattern in neural networks is a cascade of modules placed back to back, for example Linear->ReLU->Linear->ReLU. The forward method of this module chains together the series of modules passed into its constructor and its update method backpropagates the gradients and updates the weights and biases of linear layers.

2.2.4 MSE

The mean squared error (MSE) loss measures the squared distance between predicted values and targets. It is the criterion the model is optimized for. During training, the loss itself is not needed to be computed since only the gradient of MSE is necessary to update the network. Therefore, to speed things up, the MSE module's forward method simply stores the predictions and target values and does not actually calculate the loss. Upon completion of training, it is important to view the loss to gauge model's capabilities, and an alternative forward method, named __forward is provided to actually calculate the loss when desired.

2.2.5 Training

The training module accepts a Sequential module, an input, a target, and some hyperparameters, and performs gradient descent to update the model [11].

3. EXPERIMENT SETUP

The primary objective of this experiment is to compare the performance of different grid and block configurations on optimized and unoptimized CUDA implementations of neural network training, especially the linear layers. The study aims to quantify the effect of various CUDA programming optimizations, such as improved thread and block mapping, and cache aware computations, on the overall training time and efficiency. The experiment also features CPU-based implementations as baselines. All experiment results are averaged over 5 runs.

3.1 System Setup

All experiments were conducted on a local machine with the following hardware and software configurations:

- CPU: Intel Core i5-9300H (4 cores, 8 threads)
- GPU: NVIDIA GeForce GTX 1660 Ti (6 GB GDDR6 VRAM)
- System Memory: 16 GB DDR4 RAM
- CUDA Version: 12.6
- Operating System: Windows 11 (64-bit)

The GTX 1660 Ti is a Turing-architecture GPU featuring 1,536 CUDA cores and 192-bit memory bandwidth. All GPU kernels were compiled using the NVIDIA CUDA compiler nvcc under default optimization settings.

3.2 Workloads

There are two distinct workload configurations that were designed to test different performance bottlenecks:

1) Small Batch, Large Network
   - Batch size: 1,000
   - Model Parameters: 12.5 million
   - Characteristics: Compute-intensive workload that emphasizes arithmetic throughput and efficient thread scheduling.
2) Large Batch, Small Network
   - Batch size: 10,000
   - Model Parameters: 125.5 thousand
   - Characteristics: Memory-intensive workload with frequent global memory access and limited arithmetic intensity.

3.3 Implementations

Three different implementations of neural networks were developed for comparison. All three implementations are evaluated across both workloads.

3.3.1 CPU (Baseline)

This implementation is executed entirely on the CPU using standard C++ loops and libraries. It serves as a reference to measure the benefits of GPU acceleration and parallel processing. The entire code is executed on a single CPU thread sequentially.

3.3.2 Unoptimized CUDA Implementation

This is a naïve implementation of neural networks in CUDA. It is a direct GPU translation of the CPU code with minimal optimizations. It uses default thread and block configurations for all kernels (forward, backward, update) with no consideration for occupancy or memory access patterns. This means every kernel runs with the same grid shape and dimensions even though their data sizes differ. As a result, some kernels may launch too many or too few threads, leading to lower GPU utilization. It can be used as a baseline to measure the effects of CUDA-level optimizations.

3.3.3 Optimized CUDA Implementation

This implementation is an enhanced version of the unoptimized version. It focuses on maximizing GPU utilization with cache-aware loop structures to improve data reuse and minimize redundant memory access. It also uses __restrict__ pointers to help the compiler optimize memory access. It has a more adaptive computation of grid and block dimensions for different kernel types. Each kernel adapts its grid to match the data it processes:

1) Forward: based on (bs, n_out)
2) Backward: based on (bs, n_in)
3) Update: based on (n_in, n_out)

4. RESULTS AND ANALYSIS

4.1 Small Batch, Large Network

For a compute-intensive workload, the CPU runtime averaged to 2938.77s. This is a significant amount of computing time for a task of this scale. The best runtime in Unoptimized CUDA implementation was 5.445s (540 times faster than CPU) for block sizes 8x8 and grid size 0x0 - 0x0 grid size represents an adaptive grid size to cover all the blocks in a single grid - and the worst runtime was 85s for block size 128x4 and grid size 8x8, which is still more than 34 times faster than CPU. The best runtime for optimized CUDA implementation was 3.7s (1.5 times faster than unoptimized version) for block size 16x16 and grid size 0x0 and the worst runtime was 14.467s (6 times faster than unoptimized version). As per raw performance, adaptive grid size on optimized CUDA implementation shows unprecedented boost over normal CPU execution time. These findings are consistent with other studies that benchmark GPU acceleration for neural network training [12].

As per **Figure 2** and **Figure 3,** the runtime varies dramatically – up to 4x difference between the best and worst grid configurations for both the optimized and unoptimized implementations. This emphasizes the importance of choosing the correct configuration based on application for maximum performance. In both implementations, smaller grid sizes like 8x8 and 16x16 exhibit slower runtimes for all block sizes. This is because smaller grid sizes result in a large number of grids per kernel. These grids are executed sequentially, one after another. This results in low parallelism and very slow sequential execution. However, when the grid sizes are too large, they are slightly slower because of tail latency effect, scheduling overhead and high kernel launch costs. Moderate grid sizes perform comparatively better. Largest grid size 0x0 where there is only one grid is fastest for all cases. This is because it maximizes GPU utilization as it has all the required block in the same grid, and therefore there is always a block available for execution for the SMs upon completing the execution of previous block. However, if the blocks are not in the same grid, if one SM completes execution of a block, and there are no more blocks left in the grid, it has to wait until other SMs complete executing their blocks and the next grid in sequence is launched. We can conclude that we would prefer a single grid just large enough to cover all the blocks in a kernel for optimal performance.

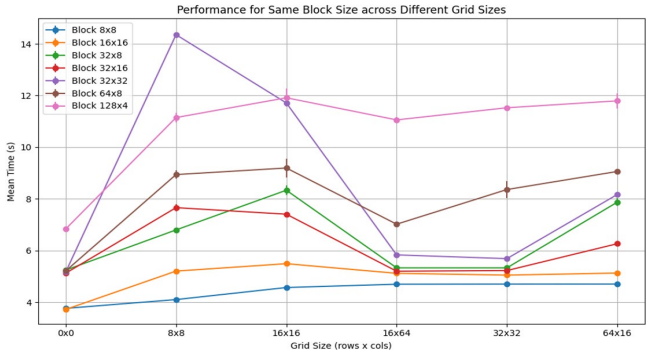

Figure 2. Performance of same block size across different grid sizes for optimized CUDA implementation.

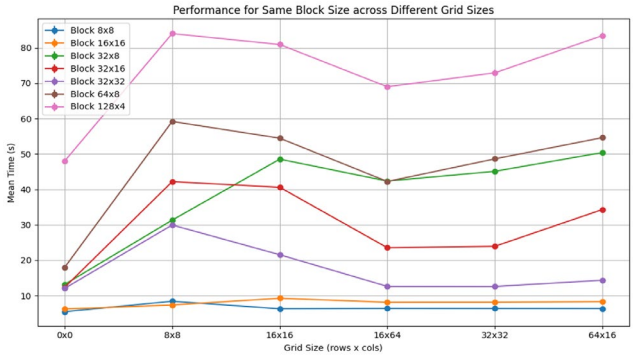

Figure 3. Performance of same block size across different grid sizes for unoptimized CUDA implementation.

It is also evident from **Figure 2** and **Figure 3** that smaller block sizes is resulting in better overall performance across all grid sizes. This is because large block sizes, which have many threads, need registers for accumulators, loop indices, pointers, etc. This increases the register pressure per SM. As a result, SMs can't keep many threads resident, thus reducing the occupancy. This reduces the latency hiding capabilities and leads to more stalls. Another reason could be that if you launch too many threads or blocks, they start touching different regions of memory at once, and cache lines get evicted before reuse. So, the ratio of computation/memory access drops. As per **Figure 4** and **Figure 5,** runtimes increase with increase in block size. For the optimized CUDA implementation, 32x32 block size has maximum number of threads -1024 threads – and thus has the highest runtime. When block size is large, each thread accesses inp[row*n_in + j] repeatedly across non-contiguous memory, i.e., the thread accesses data farther away in memory, leading to uncoalesced memory accesses. Smaller, balanced blocks minimize per-thread cache thrashing, improve memory locality and lead to high occupancy in SM. Interestingly, **Figure 5,** for unoptimized CUDA implementation sees a dip for 32x32 block size. Since this implementation uses the same block shape for all kernels, square blocks compensate and distribute workloads better leading to faster runtimes compared to rectangular blocks. Rectangular blocks amplify the issues in unoptimized implementation. Square blocks like 8x8, 16x16 and 32x32 perform better in unoptimized version compared to rectangular blocks. It is evident that smaller, square block sizes are preferable for optimal performance in both optimized and unoptimized CUDA implementations.

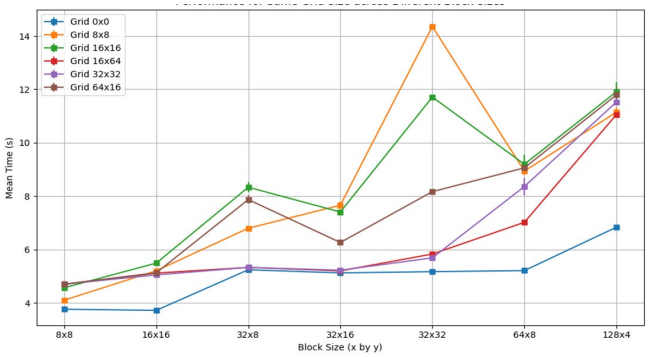

Figure 4. Performance of same grid size across different block sizes for optimized CUDA implementation.

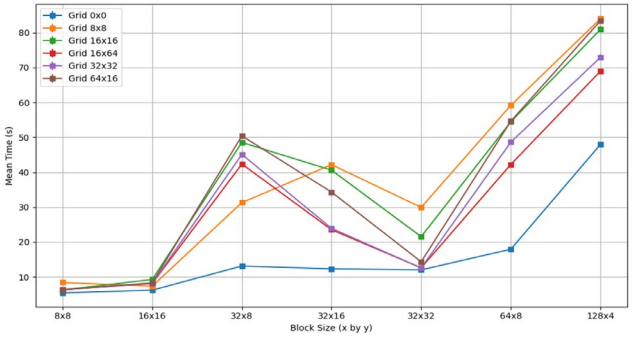

Figure 5. Performance of same grid size across different block sizes for unoptimized CUDA implementation.

We can observe similar patterns in **Figure 6** and **Figure 7** which compare the performance of different block sizes and shapes with same number of total threads. The runtime grows steadily as total threads in a block increase. 64 thread blocks (8x8) are fastest, followed by 256-thread and 512-thread blocks being slower, and 1024-thread (32x32) being the slowest. In compute-heavy kernel, more threads per block means more register pressure and fewer concurrent warps per SM leading to reduced occupancy and hence slower runtimes. For unoptimized version in **Figure 7,** 32x32 block is performing better than blocks with less threads due to the square shape that balances the performance for all three kernels, whereas skewed, narrow or long rectangular blocks amplify the effects of ill-design choices in this implementation and show huge difference in runtime. Similar patterns are also observed in optimized implementation in **Figure 6**, however the difference in runtimes is not too large. This again reinforces our conclusion to use smaller, square shaped blocks for better overall performance.

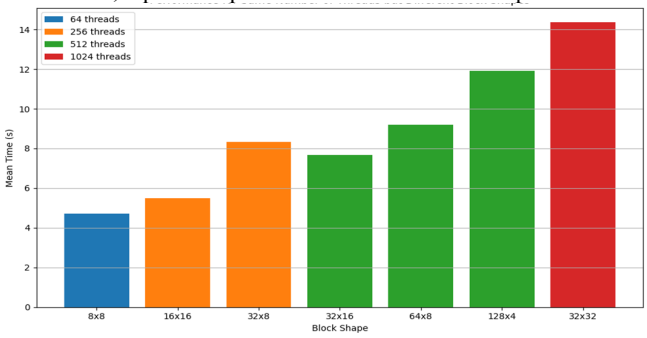

Figure 6. Performance for same number of threads but different block shapes for optimized CUDA implementation.

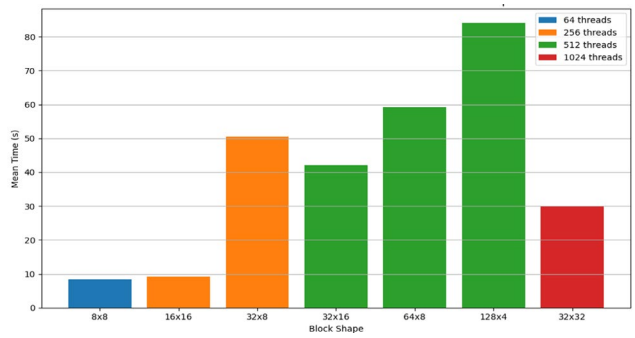

Figure 7. Performance for same number of threads but different block shapes for unoptimized CUDA implementation

Figure 8 and **Figure 9** show a heatmap of mean training times for all the configurations for optimized and unoptimized CUDA implementations, respectively. Fastest runtimes are observed for smaller block size and 0x0 grid size, i.e., a single grid that is just large enough of cover all the blocks of the kernel. On the other hand, larger block sizes with narrow or wide rectangular shapes perform poorly in both implementations.

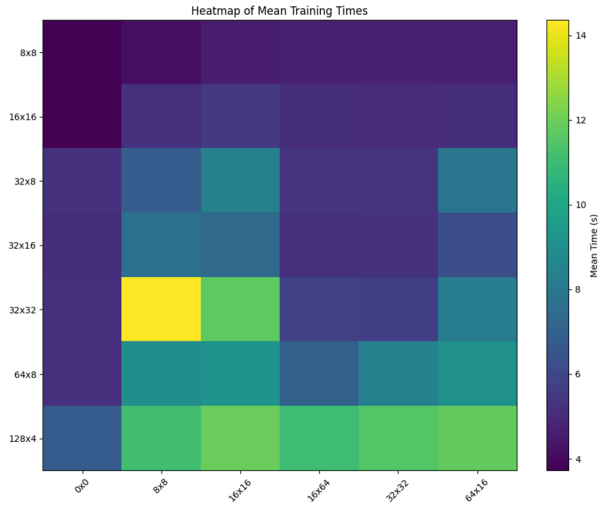

Figure 8. Heatmap of mean training times for optimized CUDA implementation.

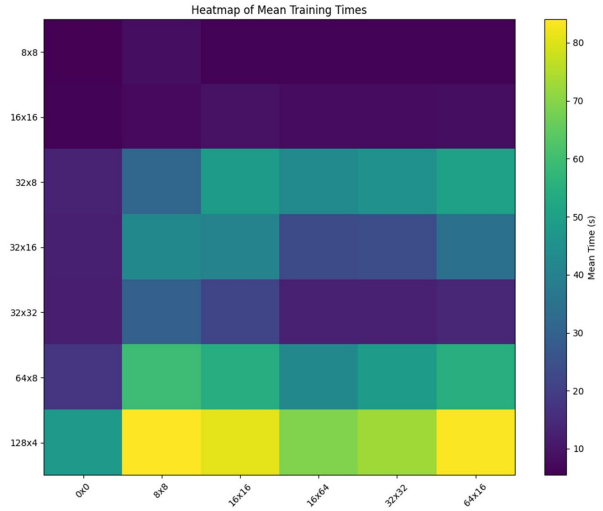

Figure 9. Heatmap of mean training times for unoptimized CUDA implementation.

4.2 Large Batch, Small Network

For this memory-intensive workload, the CPU runtime averaged to 87s. This is a significant amount of computing time for a task of this scale. The best runtime in unoptimized CUDA implementation was 1.975s (45 times faster than CPU) for block sizes 16x16 and grid size 0x0 and the worst runtime was 7.528s for block size 128x4 and grid size 16x16, which is still more than 11 times faster than CPU. The best runtime for optimized CUDA implementation was 1.835s (1.07 times faster than unoptimized version) for block size 16x16 and grid size 0x0 and the worst runtime was 2.37s (3 times faster than unoptimized version). As per raw performance, adaptive grid size on optimized CUDA implementation shows unprecedented boost over normal CPU execution time.

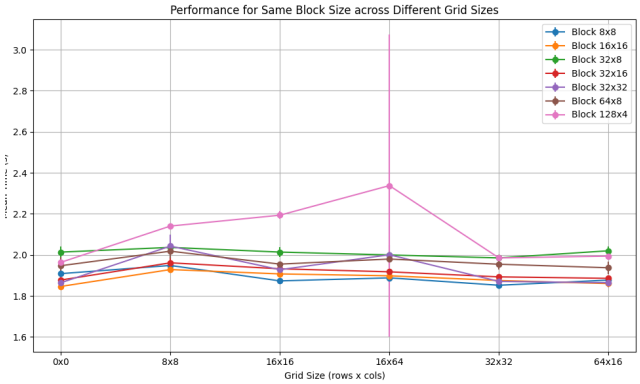

Figure 10. Performance of same block size across different grid sizes for optimized CUDA implementation.

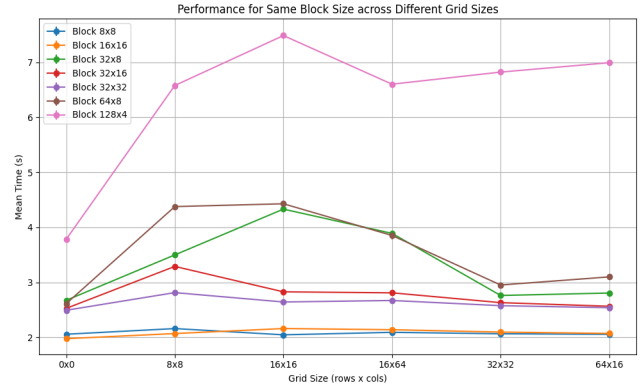

Figure 11. Performance of same block size across different grid sizes for unoptimized CUDA implementation.

As per **Figure 10** and **Figure 11,** the runtime does not vary dramatically and usually in the 1.8 to 2s range for optimized CUDA implementation. The runtime is stable for smaller block sizes in unoptimized implementation as well. Changing the block sizes or grid sizes has minimal effect -the curves are almost flat. This shows that the process is not compute bound, but memory bound in these cases. Similar observations have been reported in GPU optimization literature, where performance is limited by global memory access patterns rather than compute throughput [13]. In **Figure 11,** Larger block sizes in unoptimized implementation still exhibit the same patterns of low parallelism and high kernel launch costs seen in previous workload in **Section 4.1**. Largest grid size 0x0 where there is only one grid is fastest for all cases here as well. This is because it maximizes GPU utilization as it has all the required block in the same grid, and therefore there is always a block available for execution for the SMs upon completing the execution of previous block. 128x4 block size being skewed again shows sub-optimal performance in both optimized and unoptimized implementations, further reinforcing our conclusion of using smaller, square blocks for optimal performance.

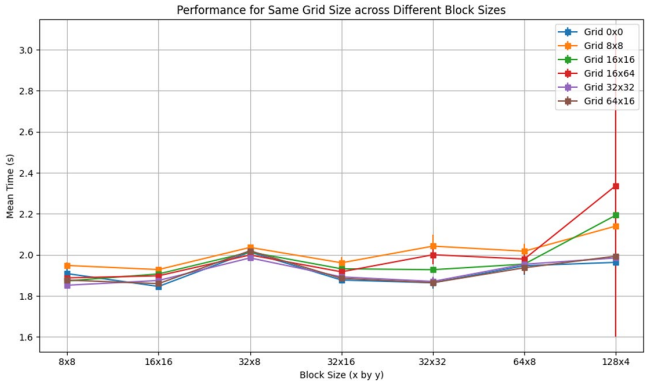

Figure 12. Performance of same grid size across different block sizes for optimized CUDA implementation.

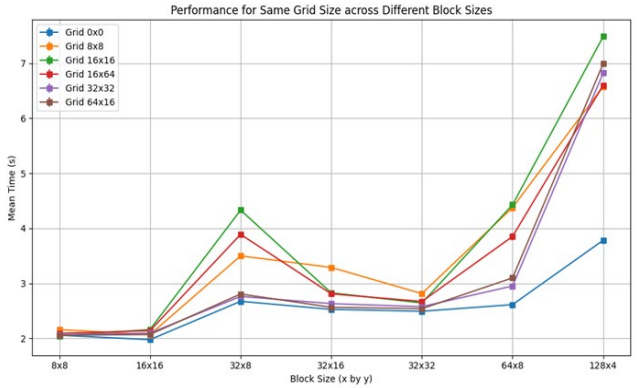

Figure 13. Performance of same grid size across different block sizes for unoptimized CUDA implementation.

As per **Figure 12** and **Figure 13,** runtimes still increase gradually with increase in block size. While the curves are fairly flat for the optimized implementation in **Figure 12,** but there is still some increase in runtime with increasing block size. Interestingly, **Figure 13,** for unoptimized CUDA implementation shows the same patterns as seen in **Figure 5.** Performance depends on warp efficiency, i.e., being able to fit more threads/blocks in the SM at a time, on memory cache reuse, and accessing data elements closer in memory, by using smaller block sizes.128x4 block creates warps spanning multiple rows with uncoalesced accesses.

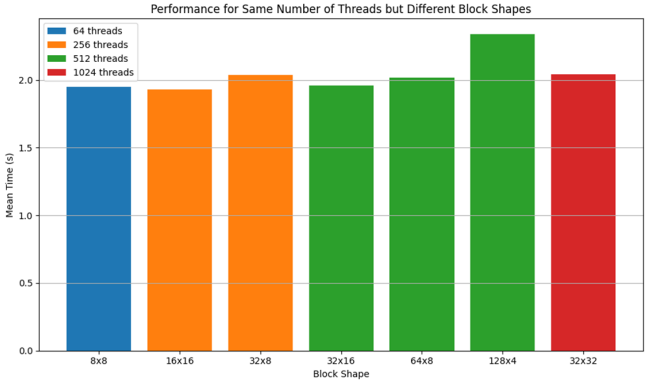

Figure 14. Performance for same number of threads but different block shapes for optimized CUDA implementation.

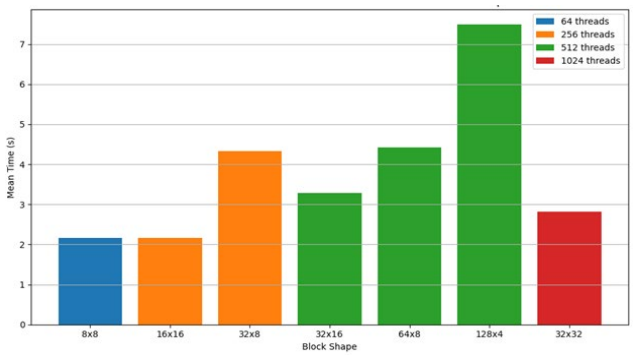

Figure 15. Performance for same number of threads but different block shapes for unoptimized CUDA implementation

We can observe similar patterns in **Figure 14** and **Figure 15** which compare the performance of different block sizes and shapes with same number of total threads. The runtime grows steadily as total threads in a block increase. While the runtimes for optimized implementation are similar, they still exhibit these patterns at some level. 64 thread blocks are fastest, followed by 256-thread and 512-thread blocks being slower, and 1024-thread being the slowest. The patterns are even more amplified for unoptimized version in **Figure 15.** 32x32 block performs better than blocks with less threads due to the square shape that balances the performance for all three kernels, whereas skewed, narrow or long rectangular blocks are significantly slower, just like the case in **Section 4.1**. This again reinforces our conclusion to use smaller, square shaped blocks for better overall performance.

Figure 16 and **Figure 17** show that performance remains consistent across most block and grid configurations, with a few spikes observed for **s**kewed block shapes such as 128x4. This indicates that the kernel is primarily memory-bound, with execution time governed by global memory access efficiency rather than computation. The performance drops for skewed blocks arise from memory coalescing inefficiency. When threads access non-contiguous memory regions due to unbalanced block dimensions, global memory transactions become fragmented, increasing latency. In contrast, balanced configurations maintain coalesced access and achieve smoother, more consistent performance.

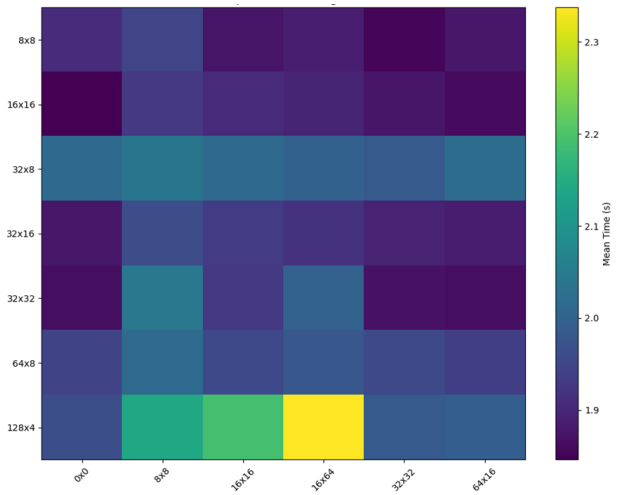

Figure 16. Heatmap of mean training times for optimized CUDA implementation.

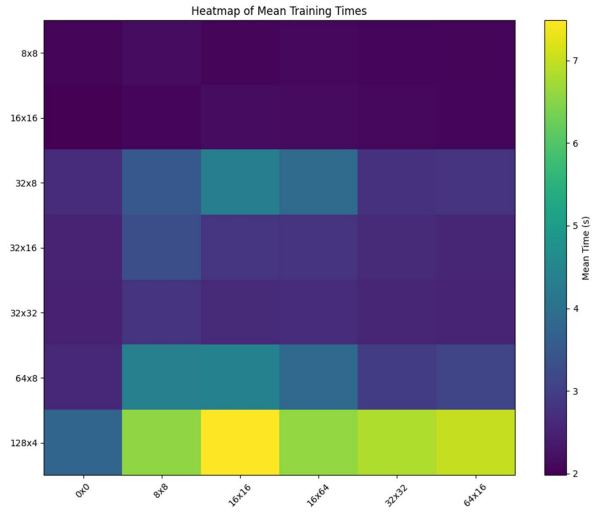

Figure 17. Heatmap of mean training times for unoptimized CUDA implementation.

5. CONCLUSION

This study demonstrated the significant performance benefits of CUDA acceleration over CPU-based execution for both compute-intensive and memory-intensive neural network workloads. Across all experiments, the CUDA implementations, both unoptimized and optimized, consistently outperformed the CPU baseline by several orders of magnitude. The optimized implementation achieved additional speedups through adaptive grid dimensioning and minor kernel-level enhancements. The results confirm that block and grid configurations have a direct and substantial impact on runtime performance. In both workloads, smaller and square-shaped blocks (e.g., 8x8, 16x16) consistently yielded the best performance across all grid sizes. These configurations provide balanced warp utilization, reduced register pressure, and efficient memory coalescing. In contrast, skewed or elongated blocks (e.g., 128x4) introduced memory coalescing inefficiencies due to unaligned thread access patterns, resulting in increased latency and reduced throughput. For grid configuration, the largest adaptive grid size (0x0) which covers all blocks within a single grid consistently produced the fastest execution. This configuration maximizes GPU utilization by ensuring all Streaming Multiprocessors remain active without waiting for additional grid launches, thereby minimizing scheduling overhead and tail latency. Conversely, excessively small or fragmented grids exhibited higher kernel launch overhead and lower parallel efficiency.

Overall, the findings emphasize that optimal CUDA performance depends on aligning block and grid configurations with the kernel's memory access and loop patterns. Implementations should use balanced, moderately sized 2D blocks and a single grid large enough to cover all blocks, ensuring efficient resource utilization, high occupancy, and coalesced memory access. Future optimizations may further benefit from tailoring kernel loop structures and memory layouts to specific workload characteristics, improving both compute throughput and memory efficiency.

The findings and performance trends presented in this study primarily apply to applications using unified memory and characterized by high register usage. In such cases, smaller or medium-sized block configurations provide higher occupancy and more stable performance. However, for workloads involving heavy shared memory reuse or prefetched data already resident on the GPU, larger block sizes may yield superior performance due to improved data locality and reduced global memory transactions

6. REFERENCES

[1] LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *Nature* 521, 436–444 (2015). https://doi.org/10.1038/nature14539.

[2] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. A. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in Advances in Neural Information Processing Systems, vol. 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," Queue, vol. 6, no. 2, pp. 40–53, Mar. 2008, doi: 10.1145/1365490.1365500.

[4] NVIDIA, "CUDA Toolkit – Free Tools and Training," NVIDIA Developer. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[5] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," arXiv preprint arXiv:1410.0759, 2014.

[6] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, 2010.

[7] D. B. Kirk and W. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 4th ed. Morgan Kaufmann, 2022.

[8] NVIDIA, CUDA C Programming Guide, NVIDIA Corp., 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[9] S. Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Morgan Kaufmann, 2012.

[10] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," Nature, vol. 323, no. 6088, pp. 533–536, Oct. 1986, doi: 10.1038/323533a0.

[12] S. Mittal, "A survey on optimized implementation of deep learning models on GPUs," Journal of Systems Architecture, vol. 99, p. 101635, 2019, doi: 10.1016/j.sysarc.2019.101635.

[13] N. C. Crago, M. Stephenson, and S. W. Keckler, "Exposing memory access patterns to improve instruction and memory-level parallelism on GPUs," ACM Transactions on Architecture and Code Optimization (TACO), vol. 15, no. 4, Art. 45, Oct. 2018, doi: 10.1145/3280851.

Columns on Last Page Should Be Made As Close As Possible to Equal Length