Enhancing CNN Performance: A CUDA Parallelization Approach

Abstract

This research investigates the substantial performance enhancements achievable through the utilization of Compute Unified Device Architecture(CUDA). CUDA is a technology designed to accelerate computations using graphics processing units (GPUs). Our primary focus is demonstrating the performance gains made possible by harnessing GPU acceleration compared to traditional CPU-based implementations. Through the development of specialized convolution and pooling algorithms tailored for CUDA execution, we have achieved a remarkable reduction in overall training time for the same CNN algorithm. The GPU version execution time decreased from approximately 37,000 seconds to a mere 500 seconds. This wide reduction represents a staggering 98% improvement in computational efficiency. Our findings underscore the impact of parallel processing and CUDA in revolutionizing the speed and efficiency of CNNs. thus demonstrating why it has been the go to solution for high-performance computing and machine learning acceleration.

Introduction

Convolutional neural networks (CNNs) are the most common neural networks for fields such as image classification, object detection, and natural language processing. However, they require substantial computational power, particularly when dealing with large-scale datasets and complex architectures. Implementing such CNNs on central processing units (CPUs) often struggle to meet the performance requirements imposed by real-world applications. Hence most CNNs leverage GPUs and parallel processing techniques to accelerate CNN training and inference. CUDA, developed by NVIDIA, is the most used framework for making use of the computational capabilities of GPUs, enabling researchers to achieve significant speedups in deep learning tasks. By distributing computations across thousands of parallel processing cores, CUDA allows for efficient execution of CNN operations. This in turn reduces processing time and enhances overall performance.

Our primary objective in this research is to demonstrate the efficacy of CUDA in improving the execution time of CNNs leveraging parallel computing for convolution and pooling layers. Through the development of specialized algorithms and optimization techniques suited for GPU execution, we aim to showcase substantial reductions in processing time and resource utilization compared to the CPU-based implementation.

Related Work

The utilization of parallel processing and GPU acceleration in deep learning, particularly within the domain of CNNs, has garnered significant attention from researchers in recent years. This section provides an overview of the existing literature and key advancements in this field.

Several studies have explored the potential of GPU acceleration in accelerating CNN computations. Krizhevsky et al. (2012) demonstrated the effectiveness of GPU-based training in their work on ImageNet classification, achieving remarkable performance gains compared to CPU-based implementations. Their work laid the foundation for future research efforts aimed at exploiting the computational power of GPUs for deep learning tasks.

Further advancements in GPU-accelerated CNNs have been facilitated by frameworks such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019), which provide high-level abstractions for GPU programming and optimization. These frameworks enable researchers and practitioners to seamlessly deploy CNN models on GPUs, leveraging sophisticated optimizations for improved performance and scalability.

In addition to high-level frameworks, researchers have developed specialized GPU-accelerated libraries and toolkits tailored for deep learning tasks. CuDNN (Chetlur et al., 2014), for example, provides optimized implementations of CNN primitives for NVIDIA GPUs, allowing for efficient execution of convolution, pooling, and other operations commonly used in CNN architectures. Similarly, cuBLAS (NVIDIA Corporation, 2020) offers GPU-accelerated linear algebra routines that can significantly enhance the performance of matrix-based computations in deep learning models.

Choi & Lee, 2017 made a CNN that ran on CUDA which had about 50% faster training time when compared to the CPU version. Jordà et al., 2022 proposed an implementation for a convolution operation on CUDA with speedup of up to 2.29***x*** with respect to the best implementation of cuda in cuDNN.

Table. 1 below summarize the main results of some of the existing works about making a CNN run faster by leveraging CUDA parallelism.

	Choi & Lee, 2017	Jordà et al., 2022
Training time reduction (%)	70	60

Table. 1 Training time reduction for related work

In summary, the existing body of work underscores the transformative impact of GPU acceleration in accelerating CNN computations and advancing the state-of-the-art in deep learning. By leveraging parallel processing techniques and specialized GPU architectures, researchers have achieved significant improvements in computational efficiency, paving the way for faster and more scalable deep learning solutions.

Parallel Computing Overview

Parallel computing is a type of computation that involves executing multiple tasks or processes simultaneously to solve a single problem. It aims to improve computational efficiency by breaking down a task into smaller subtasks that can be processed concurrently. There are several different approaches for parallel computing. We will explore some of the key approaches.

Shared Memory Parallelism

Shared Memory Parallelism is a parallel computing approach where multiple processing units share a common memory space. This enables them to access and manipulate data concurrently. This method facilitates parallel execution by dividing the workload among the processing units, thus allowing them to operate on different parts of the same data simultaneously. Multithreading is one of the most common implementations of shared memory parallelism. Multithreading allows threads to execute independently while accessing shared resources. References such as Quinn's "Parallel Programming in C with MPI and OpenMP" and Chandra et al.'s "Parallel Programming in OpenMP '' provides a detailed dive into shared memory parallelism techniques. These include practical examples and best practices for utilizing parallelism effectively in programming languages like C and C++ with OpenMP directives.

Distributed Memory Parallelism

Distributed Memory Parallelism involves multiple independent processing units. Each of these units has its own memory and communicates with each other by passing messages over a network. This breaks down computational tasks into smaller portions and distributes them among processing units for simultaneous execution. Message Passing Interface (MPI) is a widely-used library for implementing distributed memory parallel programming. Gropp, Lusk, and Skjellum's "Using MPI: Portable Parallel Programming with the Message-Passing Interface" and Pacheco's "Parallel Programming with MPI" offer comprehensive guidance on leveraging MPI for distributed memory parallelism.

Task Parallelism

Task Parallelism involves breaking down a computational task into smaller, independent tasks that can be executed concurrently. Unlike data parallelism, where the same operation is performed on different data sets, task parallelism entails executing different operations simultaneously. This approach is particularly useful for applications with diverse computational requirements or when tasks have varying execution times. Task-based parallelism frameworks, such as Intel Threading Building Blocks (TBB) or Cilk Plus, provide abstractions and tools to manage task parallelism efficiently. Reinders' "Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism" and Frigo, Leiserson, and Randall's "The Implementation of the Cilk-5 Multithreaded Language" offer good insights into this technique.

GPU Computing

GPUs are highly parallel processors designed to handle multiple tasks simultaneously. Originally developed for rendering graphics, GPUs have become popular for general-purpose parallel computing due to their massive parallelism and high computational throughput. CUDA by NVIDIA is a popular framework for programming GPUs, providing developers with tools and libraries to exploit GPU parallelism efficiently. Kirk and Hwu's "Programming Massively Parallel Processors: A Hands-on Approach" and Sanders and Kandrot's "CUDA by Example: An Introduction to General-Purpose GPU Programming" serve as a good introduction to understanding GPU's computing.

Methodology

We begin by selecting a suitable convolutional neural network (CNN) architecture for experimentation. In this study, we opt for a rather basic architecture:

- 1 convolution layer
- 1 Max pooling layer
- 1 Fully Connected layer

Since this architecture is not that complex, it's easier to show the performance gains of CUDA parallel processing.

We develop specialized algorithms and implementations of convolution and pooling operations tailored for execution on NVIDIA GPUs using CUDA. These implementations leverage parallel processing techniques to distribute computations across multiple CUDA cores, thereby exploiting the computational power of the GPU.

First, here are some CUDA Terminologies:
Kernel: Function executed on the GPU
Thread: a single GPU core
When a kernel is launched in Cuda it's executed on each thread within the launch configuration. Threads are launched in a grid that contains blocks of threads arranged in up to 3 dimensions. Each block contains threads also arranged in up to 3 dimensions as illustrated in Fig. 1.
For simplicity we will refer to convolutions kernels as filters to avoid confusion with cuda kernels.

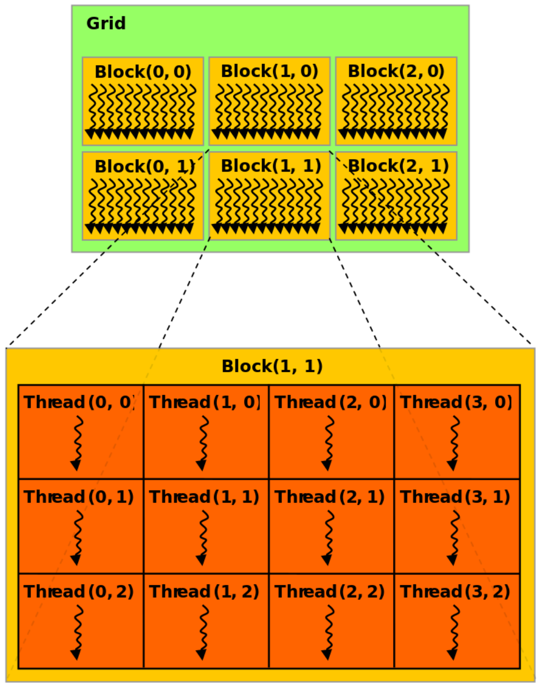

Fig.1 Thread Hierarchy in CUDA Programming

Model Architecture

The model has a simple architecture as stated above, and the convolution layer as well as the max pooling layer are CUDA optimized. Fig. 2 shows a simple overview of this architecture when it's running on the CPU and Fig. 3 illustrates the architecture when adapted for CUDA.

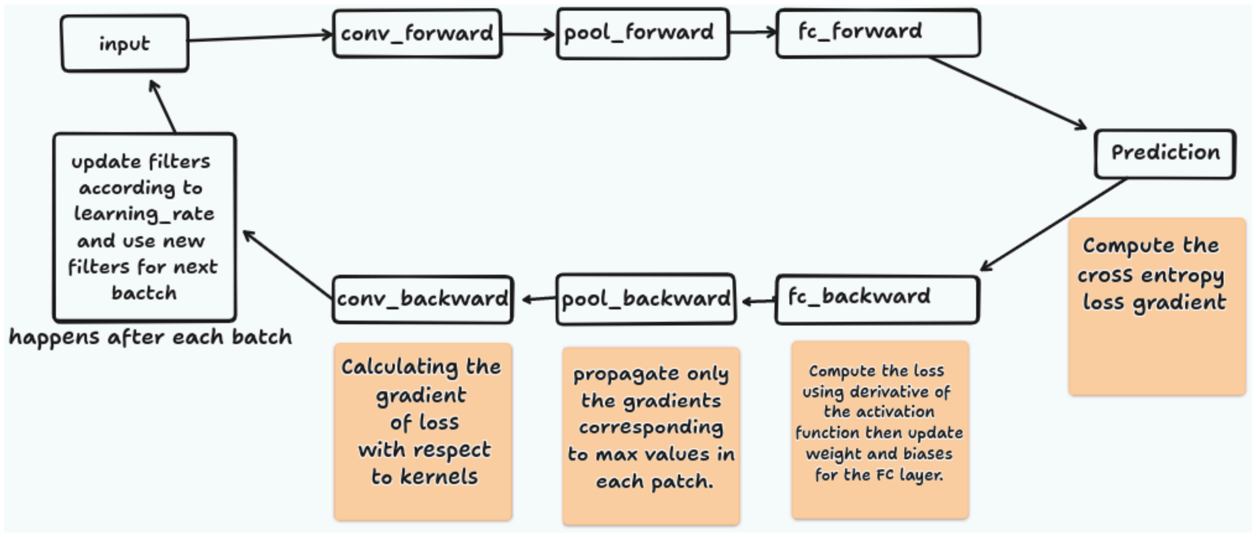
input
conv_forward
pool_forward
fc_forward
Prediction
Compute the cross entropy loss gradient
fc_backward
pool_backward
conv_backward
update filters according to learning_rate and use new filters for next bactch
happens after each batch
Calculating the gradient of loss with respect to kernels
propagate only the gradients corresponding to max values in each patch.
Compute the loss using derivative of the activation function then update weight and biases for the FC layer.

Fig. 2 CNN model architecture

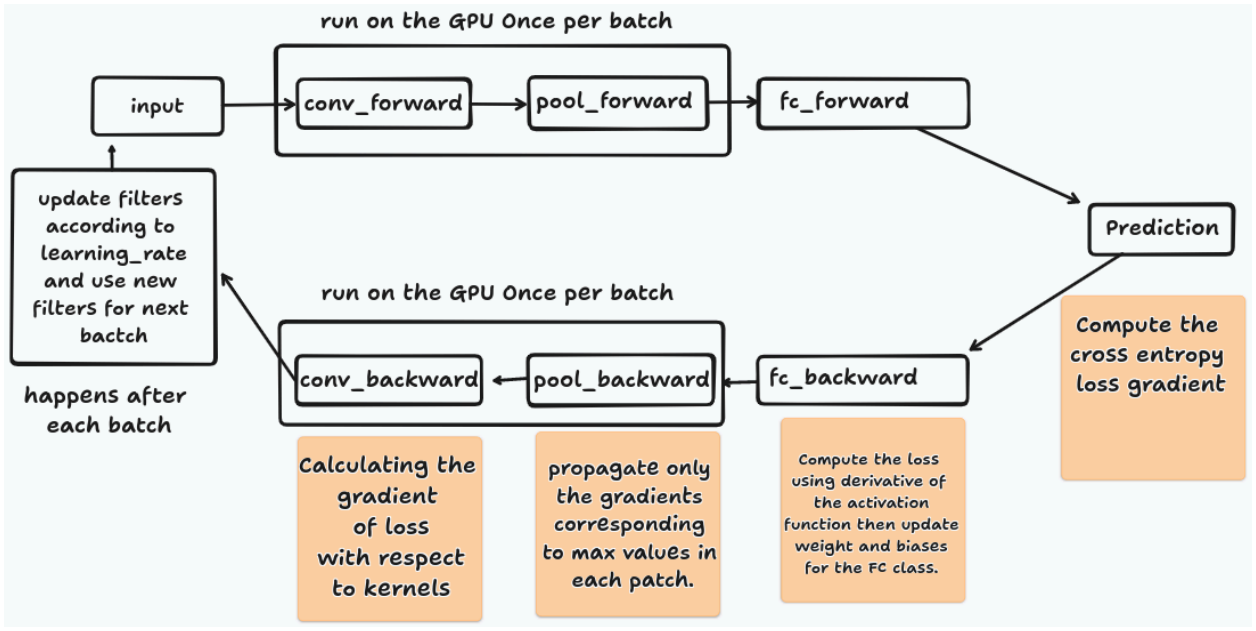
run on the GPU Once per batch
input
conv_forward
pool_forward
fc_forward
Prediction
Compute the cross entropy loss gradient
run on the GPU Once per batch
fc_backward
pool_backward
conv_backward
update filters according to learning_rate and use new filters for next bactch
happens after each batch
Calculating the gradient of loss with respect to kernels
propagate only the gradients corresponding to max values in each patch.
Compute the loss using derivative of the activation function then update weight and biases for the FC class.

Fig. 3 CNN model architecture for CUDA

The model when run on the CPU sequentially executes every layer for every input but the filters are not updated. This is to ease comparison with the CUDA optimized version as filters update only occurs once per batch for that version. In Fig. 3 we notice that convolution and pooling layers are launched as CUDA kernels once per batch as a forward pass and once as a backward pass to compute gradients.

Convolution Layer

The convolution layer produces a feature map through a convolution operation between the input image and all the filters. In the rest of this paper arrays dimensions will be given as follows (N,C,H,W) which stands for number of items, channels,height and width of each item. Assuming N is the batch size, we have a shape of (N,1,28,28) for the input and (8,1,3,3) for the filters. The Z dimension matches the channels (C).

Output Size

The output size can be calculated with the following formula:

- $outputX : inputX - filterX + 1$
- $outputY : inputY - filterY + 1$
- $outputZ : inputZ - filterZ + 1$

With this we get a shape of (N, 8, 26, 26) for the output of the convolution.

Block and grid size

To determine the optimal launch configuration for our kernel we need to find the number of blocks and the number of threads within those blocks in such a way that each thread within the configuration is responsible for a single value in the output of the convolution. After experimenting with various values we found that for our case a threads number of (8, 8, 8) works the best. The following formula is used to determine each of the dimensions for the blocks.

- X : $(N + threads[0] - 1) // threads[0]$
- Y : $(outputY + threads[1] - 1) // threads[1]$
- Z : $(outputX + threads[2] - 1) // threads[2]$

For N=16 we get the following configuration kernel<<blocks(2, 4, 4), threads(8, 8, 8)>>.
We can verify if the number of threads is greater than the number of distinct values in the convolution output, note that each thread handle all different channel for a single value hence total number of distinct values are $16 * 26 * 26 = 10816$ and the total number of thread are $2 * 4 * 4 * 8 * 8 * 8 = 16384$, thus clearly covering the output.

CUDA Convolution operation

Many different variations of the convolution operation exist for CUDA. Choi & Lee, 2017 run a block for each feature map's output as shown in Fig. 4. This means that each block will have the same size as the filter and the total number of blocks is equal to all the features map for all input images.

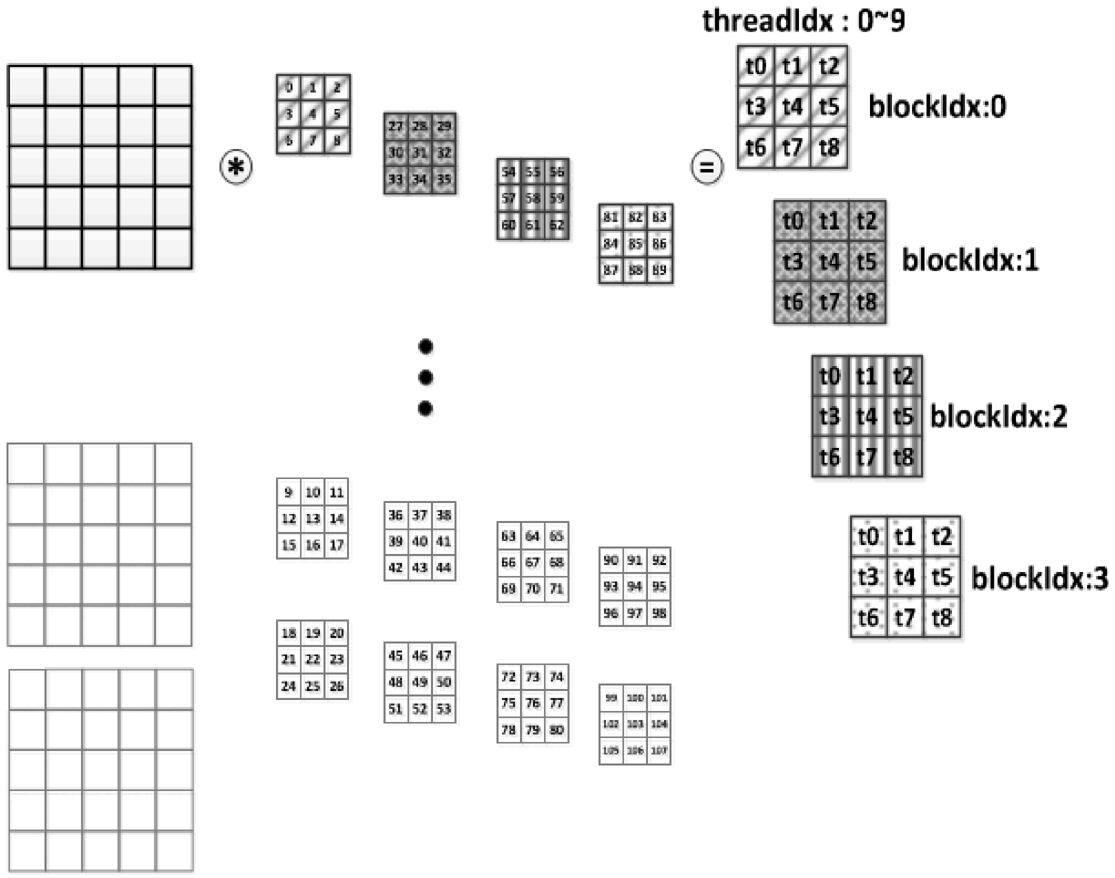

Fig. 4 Convolution operation architecture in CUDA (Choi & Lee)

One possible issue with such architecture is the amount of blocks that need to be launched which could impact the performance of the model. For this paper we went with a different architecture, the convolution operation each thread is responsible for a single value for all filters in the output. This means that a single block handles all features maps for a single input. In other words we launch one block per input. Fig. 5 illustrates the working principle of our convolution operation on CUDA for each thread.

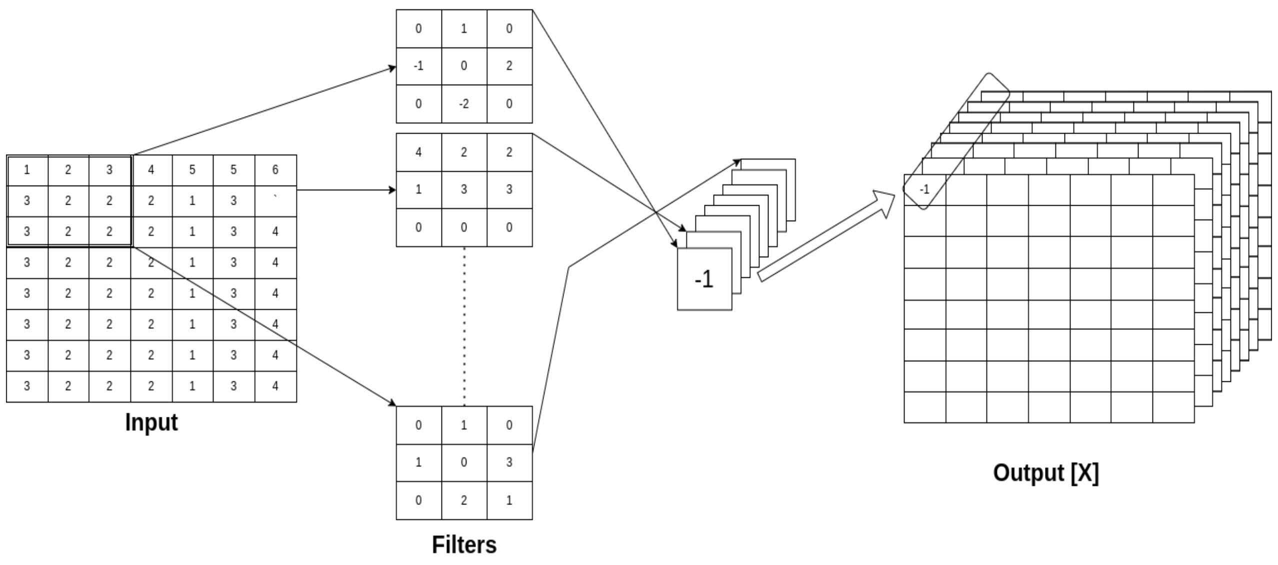

Fig. 5 Custom Convolution operation architecture for CUDA

Max Pooling Layer

The max pooling layer is quite straightforward, it just reduces the dimension of the input by the size of a given window. The input being the output of the convolution layer.

Output Size

Using a standard window size of 2x2, the output size will be divided by two giving (N, 8, 13, 13).

Block and grid size

We use the same thread configuration of (8, 8, 8) and for the block configuration we just divide the Y and Z by 2 since the output dimensions are divided by 2.

- X : $Same\ as\ Convolution$
- Y : $ConvolutionY\ //\ 2$
- Z : $ConvolutionZ\ //\ 2$

For N=16 we get the following configuration kernel<<blocks(2, 2, 2), threads(8, 8, 8)>>.

Max Pooling Operation

Max pooling on CUDA is quite straightforward, we launch a block with the size of the pooling operation result. Each thread will extract the max value for the corresponding region. Choi & Lee, 2017 run a block per input as shown in Fig. 6

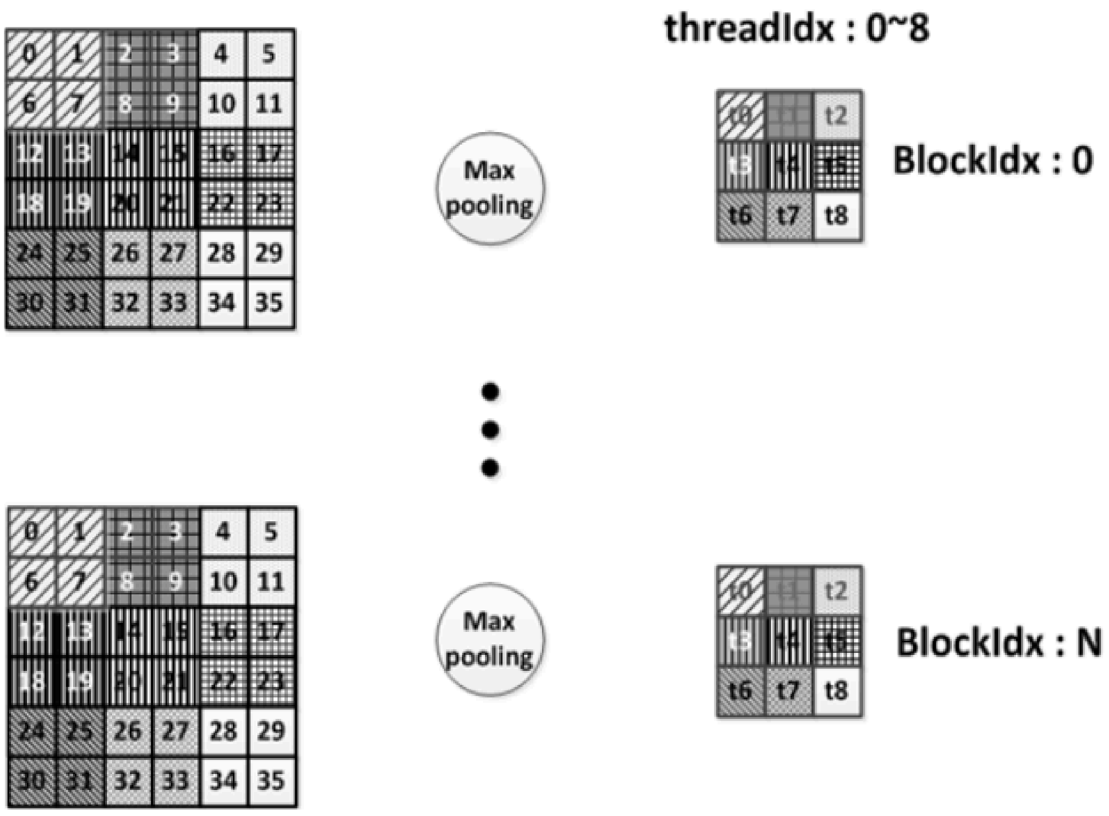

Fig. 6 Max pooling CUDA (Choi & Lee)

In our case we made a slight adjustment to allow each block to handle all channels for a given input. Fig. 7 shows an overview of our version of the max pooling operation on CUDA.

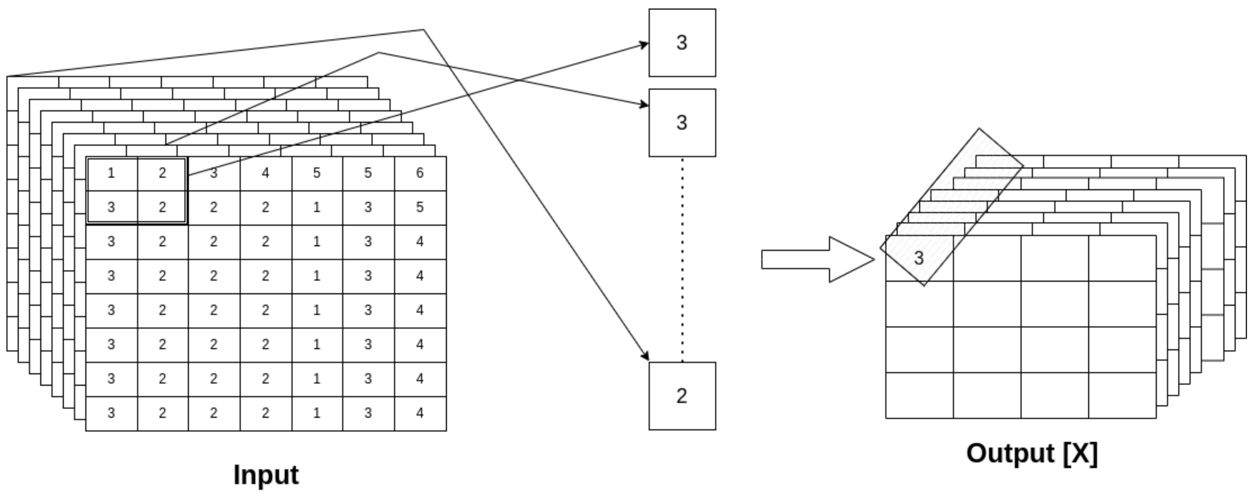

Fig. 7 Max pooling operation inside a single thread for CUDA

Fully Connected Layer

The fully connected (FC) layer is executed on the CPU so there are no blocks or threads dimensions to compute. We flatten the output of the max pooling layer and multiply it with the FC weights and add the bias. Then we use the softmax activation function to get a probability for each class and predict the higher probability class as the output.

Backward Pass & Weight Update

The backward pass is an integral part of a CNN as it allows the network to iteratively adjust its parameters and improve learning. During the backward pass, gradients are computed with respect to the loss function, in this case that function being the **cross entropy loss.** This involves propagating the error from the output layer back through the network, updating weights and biases using stochastic gradient descent(SGD). It’s important to note that this weights update using SGD occurs after each batch using the accumulated gradients for that batch.

Experimental Setup

All Experiments are conducted on a Lenovo IdeaPad Gaming 3 15IHU6 Laptop running Ubuntu 22.04 LTS as an operating system with the 8 core 11th Gen Intel® CoreTM i7-11370H CPU and the NVIDIA GeForce RTX 3050 GPU. We are running NVIDIA drivers version:545.29.02 with CUDA version 12.3 and cuDNN version 8.7.0.

For this experiment we are using cuda from the Numba python library. This allows us to write python code that will be compiled into CUDA kernels and device functions following the CUDA execution model.

For training and evaluation purposes, we utilize the **MNIST** and **Fashion_MNIST** datasets. These datasets are some of the most popular datasets to test performance of CNNs because they contain greyscale images and accuracy should always be very high. Moreover our CNN has a very basic architecture since we are focusing on training time improvements making these datasets the perfect choice for this use case.

Prior to training, the input images are preprocessed to (1, 28, 28) shape to ensure uniformity and compatibility with the CNN model.

During training, we monitor metrics such as loss, accuracy and epoch execution time to assess the model's progress and performance. To evaluate the performance of the trained model, we measure accuracy on a separate validation dataset. This provides insights into the model's classification performance and its ability to generalize to unseen data.

Results

To conduct these experiments the following hyperparameters we used :

- Learning rate: 0.01
- Batch size: 16
- Number of epochs: 20
- Number of filters: 8

Table 2 shows the best run for both the CPU and GPU while table 3 shows the average for 5 runs.

	Accuracy	Training Time (in seconds)
CPU (MNIST)	0.9463	36550
GPU (MNIST)	0.9495	516
CPU (FASHION_MNIST)	0.8358	36900
GPU (FASHION_MNIST)	0.8330	545

Table. 2 Best accuracy and training time comparison for CPU/GPU

	Accuracy	Training Time (in seconds)
CPU (MNIST)	0.9439	36700
GPU (MNIST)	0.9469	566
CPU (FASHION_MNIST)	0.8330	37100
GPU (FASHION_MNIST)	0.8298	538

Table. 3 Average accuracy and training time comparison for CPU/GPU

As demonstrated above we achieved an impressive reduction of about 98% in training time for both of these datasets without compromising the accuracy of the model. This shows how effective CUDA is in parallel processing and why it became the go to solution for operations that can be parallelized. The difference in accuracy between the two dataset could be explained by the fact that **Fashion_MNIST** dataset contains more complex shapes such as shirt and short when compared to only digits in the **MNIST** dataset. Furthermore our architecture is very basic so it can not extract complex features, by adding more layers we could allow the model to extract complex features more effectively thus increasing the overall accuracy.

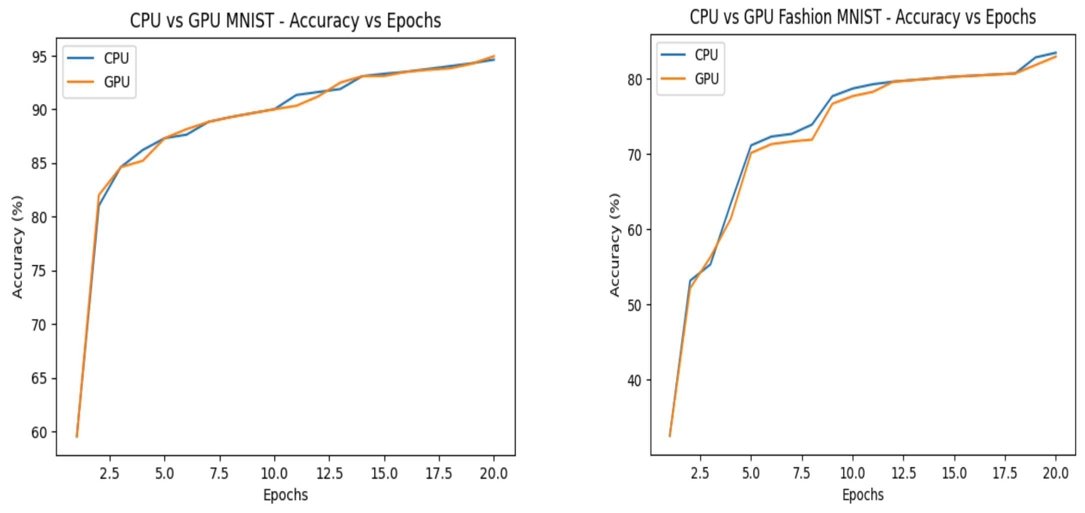

Fig. 8 Accuracy per epoch of the CPU & GPU

Fig 8 compares how the training accuracy increases for the CPU and GPU with each epoch for both the **MNIST** and the **Fashion_MNIST** datasets. This graph is inline with what is shown by tables 2 and 3, demonstrating that the only difference between the GPU and CPU is the fact that the GPU can execute the same algorithm much faster.

As mentioned above we run a kernel for each batch so in theory if we increase the batch size that will reduce the number of total batch meaning we will launch fewer kernels. Table 4 shows the effect on different batch sizes on training time and accuracy. The **MNIST** dataset was used for this but a similar result should be observed if we have used any other dataset.

	Accuracy	Training Time (in seconds)
Batch size 32	0.9374	394
Batch size 64	0.9358	314
Batch size 128	0.9304	266

Table. 4 Accuracy and training time for different batch sizes

The above result shows that when the batch increases, the training time significantly decreases while the accuracy drops slightly. This is inline with the fact that with greater batch size we launch fewer kernels hence taking less time to complete the process but also updating weights less frequently which explains the slight drop in accuracy.

Conclusion

In this paper, a method of parallel processing in training data in convolutional neural networks was proposed based on CUDA. CUDA enabled GPU platform provides to the developers a different architecture capable of highly parallel computing when compared with the traditional CPU. The allocation method of blocks and threads in the GPU was used to process the computation in parallel in a distributed manner. The experiment result showed that training time was found to be reduced by about 98% when compared to the version running on the CPU. That improvement increased to 99.3% when batch size was increased but it came with a slight drop in accuracy. This demonstrates the effectiveness of CUDA in highly parallelizable tasks and explains why CUDA has been a catalyst for deep learning algorithms in recent years.

To conclude, our study demonstrates how simple and transformative it’s to optimize your own algorithms for CUDA with amazing results. Thus offering a pathway for researchers and developers to unlock unprecedented levels of performance and efficiency in their parallelizable computational tasks.

References

Quinn, M.J. (2004). Parallel programming in C with MPI and openMP. New Delhi ; Madrid: Mcgraw-Hill Education.

Chandra, R., Menon, R., Dagum, L., Kohr, D., Dror Maydan and McDonald, J. (2000). Parallel Programming in OpenMP. Elsevier.

Gropp, W., William D.. Gropp, Lusk, E., Skjellum, A. and Fellow, D. (1999). Using MPI. MIT Press.

Pacheco, P.S. (1997). Parallel programming with MPI. San Francisco: M. Kaufmann.

Reinders, J. (2007). Intel Threading Building Blocks. ‘O’Reilly Media, Inc.’

Frigo, M., Leiserson, C. E., & Randall, K. H. (1998). The Implementation of the Cilk-5 Multithreaded Language.

Kirk, D. and Hwu, W.-M.W. (2010). Programming massively parallel processors : a hands-on approach. Amsterdam ; London: Morgan Kaufmann.

Sanders, J. and Kandrot, E. (2010). CUDA by Example : An Introduction to General-Purpose GPU Programming. Sydney: Pearson Education, Limited.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2016). TensorFlow: Large-scale machine learning on heterogeneous systems. arXiv preprint arXiv:1603.04467.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In Advances in neural information processing systems (pp. 8024-8035).

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., & Shelhamer, E. (2014). cuDNN: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759.
NVIDIA Corporation. (2020). NVIDIA cuBLAS Library Documentation. Retrieved from https://docs.nvidia.com/cuda/cublas/index.html.

Choi, S., & Lee, K. (2017a). A CUDA-based implementation of Convolutional Neural Network. 2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT). https://doi.org/10.1109/caipt.2017.8320682.

Pacheco, P. S., & Malensek, M. (2022). GPU programming with Cuda. An Introduction to Parallel Programming, 291–360. https://doi.org/10.1016/b978-0-12-804605-0.00013-0.

Jordà, M., Valero-Lara, P. & Peña, A.J. cuConv: CUDA implementation of convolution for CNN inference. Cluster Comput 25, 1459–1473 (2022). https://doi.org/10.1007/s10586-021-03494-y.

Numba for Cuda GPUs. Numba for CUDA GPUs - Numba 0+untagged.2155.g9ce83ef.dirty documentation. (n.d.). https://numba.readthedocs.io/en/stable/cuda/index.html.

Cuda Refresher: The Cuda Programming Model. NVIDIA Technical Blog. (2023, June 12). https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model.