Convolutional Neural Network Implementation on GPU

Claudio Camolese
École nationale supérieure d'informatique et de mathématiques appliquées
https://github.com/claudiocamolese/CUDA-CNN-from-scratch.git

Abstract

This report presents a from scratch implementation of a Convolutional Neural Network entirely developed in CUDA. The architecture includes two convolutional layers with ReLU activations, max pooling, and a fully connected classifier. The objective of this project is to report the training results of the same neural network on both CPU and GPU.
On the CPU side, the implementation is written in Python and leverages the PyTorch library for training. Additionally, users are given the possibility to modify parameters such as pin_memory and $num\_workers$ in order to adjust the training configuration.
On the GPU side, the implementation is written in CUDA and no external libraries are used for training. Two scenarios are analyzed: a naive implementation that serves as a baseline and an optimized version that incorporates performance improvements to accelerate the training process.
The study uses the MNIST and FASHION datasets, both consisting of 28×28 grayscale images with 10 classes. MNIST, with 60,000 training and 10,000 test images of handwritten digits, is a standard benchmark for evaluating models and training performance. FASHION, with similar structure but more visually subtle classes, presents a greater challenge for classification, often resulting in lower accuracy.
The project leads to three main conclusions. First, shifting computation from CPU to GPU provides the largest boost, with roughly an $8\times$ speedup. Second, applying algorithmic optimizations on the GPU adds a significant further improvement (about 4–5$\times$), exceeding the gains from GPU generation alone. Third, tuning the DataLoader (number of workers and pinning memory) has a comparatively minor effect relative to hardware acceleration.
The project was also tested with different thread block sizes, showing that optimal performance requires balancing thread level parallelism with hardware constraints.
The code implementation and instructions about how to run the code can be found here.

1. Introduction

Convolutional Neural Networks (CNNs) are a type of deep learning model that have become central to modern artificial intelligence, especially for tasks involving images, video, and other spatial or temporal data. Over the past few years, AI has seen rapid evolution, moving from simple machine learning algorithms to highly complex neural networks capable of achieving human-level performance in many domains, from image classification to natural language understanding. CNNs play a crucial role in this progress due to their ability to automatically extract hierarchical features from raw data, making them extremely powerful for perception tasks.

GPUs (Graphics Processing Units) have been instrumental in this AI evolution because they excel at parallel computation, allowing CNNs to process large amounts of data simultaneously. CPUs (Central Processing Units), by contrast, are optimized for sequential tasks and have far fewer cores, making them less efficient for training deep networks. Using GPUs has therefore been key to reducing training time, enabling larger and more sophisticated models and accelerating the overall advancement of AI technologies.

Parallelism in a CNN can be introduced at several levels. Convolutional and pooling operations can be executed independently across different regions of the input feature maps, while matrix multiplications in fully connected layers can also be parallelized. Additionally, computations across different channels and batches can be processed concurrently, allowing GPUs to exploit thousands of cores effectively. By identifying these opportunities for parallel execution, training and inference can be significantly accelerated, making large-scale CNNs feasible even on complex datasets.

All the experiments reported in this report were performed on the cluster using the parameters *–gres=gpu:1*, *–cpus-per-task=4*, and *–mem=2G*. If not specified, it is assumed to use a 256 `blocksize`.

The number of training epochs is set to 5, and the learning rate is kept constant throughout all epochs. In typical CNN applications, the learning rate is usually decreased during training to improve convergence and avoid over-

shooting minima. However, given the small number of epochs, the limited dataset size and the fact that achieving the highest possible accuracy is not the goal of this project, no fine-tuning of the learning rate or other training parameters is performed.
The batch size used is 64 for both Pytorch and Cuda code.

1.1. CNN architecture

The implemented CNN starts with a first convolutional layer (Conv1) that applies multiple filters to the 28×28 input images, extracting low level features such as edges, lines and simple textures. The output of this layer is passed through a ReLU activation function, which introduces non-linearity, accelerates convergence, and mitigates vanishing gradient issues.
The second convolutional layer (Conv2) takes the feature maps produced by the first layer and learns higher level representations by combining the low level patterns captured earlier, again followed by a ReLU activation to maintain non-linearity and promote efficient training.
A convolutional operation is written as:

$$h = \texttt{ReLU}(W * x + b) \tag{1}$$

where $x \in \mathbb{R}^{28\times28}$ is the input image, W and b are the filters and the bias of the current layer respectively, $*$ denotes the convolution operation and $ReLU(z) = max(0, z)$.

After the convolutional layers, a max pooling operation is applied to the feature maps, reducing their spatial dimensions while retaining the most salient information. This pooling step not only decreases the computational load for subsequent layers but also provides translational invariance, allowing the network to recognize features regardless of minor shifts in the input images.
The pooling operation is written as:

$$h^i = \texttt{MaxPool}(h^{i-1}, k) \tag{2}$$

where k is the pooling window size and the MaxPool operation takes the maximum values within each $k \times k$ window.

The pooled feature maps are then flattened into a one-dimensional vector, enabling the fully connected layer (FC) to process them. The FC layer performs reasoning, combining all the learned features to produce a raw score for each class.
The FC layer performs:

$$z = W^{FC}h + b^{FC} \tag{3}$$

where W^{FC} are the weights of the fully connected layer and b^{FC} is the bias.

Finally, the softmax layer converts these scores into probabilities, facilitating classification through cross entropy loss during training.
The softmax is computed as:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{4}$$

where $\hat{y_i}$ is the predicted probability for class i. The cross entropy loss is:

$$\mathcal{L} = -\sum_i y_i log(\hat{y}_i) \tag{5}$$

and is used when there is a classification task among multiple classes.
The total number of learnable parameters in the CNN is calculated by summing the contributions of all layers with trainable weights. The first convolutional layer has 16 filters of size 3×3 applied to a single-channel input, giving $(3\times3\times1+1)\times16 = 160$ parameters, where the +1 accounts for the bias per filter. The second convolutional layer takes the 16 feature maps from the previous layer and applies 32 filters of size 3×3, resulting in $(3\times3\times16+1)\times32 = 4,640$ parameters. After the convolutional and pooling operations, the feature maps are flattened into a vector of size 4,608 ($32 \times 12 \times 12$), which is fully connected to the 10 output classes, yielding $(4,608 + 1) \times 10 = 46,090$ parameters in the fully connected layer. The softmax layer does not introduce any additional learnable parameters. Summing all contributions, the network has a total of $50,890$ trainable parameters.

2. Pytorch implementation

For the CPU implementation, Python and the PyTorch library were used for training. PyTorch provides an option to choose the device on which the code will run, either 'cuda' or 'cpu'. Setting the device to CUDA leverages GPU acceleration, significantly reducing computation time, while using the CPU results in much longer training times.
In PyTorch, `pin_memory=True` and $num\_workers > 0$ are two important `DataLoader` options that affect data transfer and preprocessing efficiency. When `pin_memory=True` enabled, it allocates pinned (page-locked) memory on the CPU. This is only useful if the data will be copied to a GPU (device='cuda') because pinned memory allows asynchronous transfers using cudaMemcpyAsync, making CPU $\rightarrow$ GPU data copies much faster. If the device is CPU, setting `pin_memory=True` has no practical effect. In short, it is useful and essentially necessary only for GPU training. The $num\_workers > 0$ option controls the number of parallel worker processes used to load batches in the `DataLoader`. It works for both CPU and GPU training because it affects only the preprocessing

of data on the CPU. When training on GPU, data loading and preprocessing happen in parallel with the GPU computation, minimizing idle time. On CPU, multiple workers enable parallel data loading for training, though it does not accelerate GPU copies. However, the number of workers cannot be increased indefinitely. Excessive workers can overwhelm system resources (CPU cores, memory), causing the DataLoader to run slower or even freeze. PyTorch will issue a warning in this case. To make the code as user-friendly as possible, simply specifying *device cuda* or *device cpu* when running *main.py* allows the user to choose which device to use.
The purpose of this example is to demonstrate how a neural network can be built in just a few lines of code while relying on PyTorch to handle all the backend optimizations, including efficient GPU acceleration, automatic differentiation, and memory management.

3. CUDA implementation

In this section, the implementation written entirely in CUDA without the use of any auxiliary libraries is discussed. Two versions are presented: a `naive` version and an `optimized` version. The purpose of this division is to demonstrate how applying different optimization techniques can drastically improve performance.

Since in deep learning libraries vectors are initialized automatically, in the CUDA version the vectors to be updated are initialized with *He* initialization. This method initializes weights by sampling from a normal distribution $W \sim \mathcal{N}\left(0,\ \frac{2}{n}\right)$ and sets biases to zero, which mitigates vanishing and exploding gradient problems when using ReLU activations, where n is the number of input neurons. This results in faster convergence and more stable training. For both versions, the project implements a forward kernel and a backward kernel for each layer of the CNN. CUDA events are created to measure training time.

3.1. Naive version

The aim of this version is to implement a full CUDA implementation without taking care of major optimization paradigm. Each CUDA kernel is launched with the appropriate number of blocks and threads ensuring that all elements are processed efficiently and GPU cores are fully utilized. Only the current batch of images and labels is copied to the GPU per iteration, minimizing transfer overhead. All intermediate gradients remain on the GPU until they are used to update parameters.
At the beginning of execution, host memory is allocated for the training and test datasets. By default, the dataset used is MNIST, but a different dataset can be specified via a command line argument (eg. FASHION). The initialization phase allocates all necessary GPU memory using `cudaMalloc`. This includes memory for input images, labels, convolutional layer weights and biases, intermediate activations, pooling outputs and indices, flattened representations, fully connected parameters, softmax probabilities, loss values, and all gradient tensors required for backpropagation. Dimensions for weight tensors and intermediate outputs are computed using predefined constants such as channel counts, filter sizes, and output spatial dimensions.

Each kernel is designed to have one thread per output element and uses a fixed `Blocksize` and a $\texttt{gridDim} = (\text{total} + \texttt{Blocksize} - 1)/\texttt{Blocksize}$ maximizing GPU occupacy.

For each epoch, batches are processed sequentially. At the beginning of each batch, the corresponding images and labels are copied from host memory to device memory allowing parallelism within the batch images.

The forward pass starts with the first convolutional layer, where the `ConvForward` kernel computes feature maps using the input images and learned filters. The output is immediately passed through a ReLU activation using `ReLUForward`. The second convolutional layer follows the same pattern: convolution via ConvForward, then activation via ReLUForward. After the convolutional stages, max pooling is applied using `MaxPoolForward` kernel, which reduces spatial resolution and records the indices of maximum values for use during backpropagation.

The pooled output is then flattened into a one dimensional representation using `FlattenForward` kernel. This flattened tensor is passed to the fully connected layer, implemented with `FullyConnectedForward`, which produces class logits. Finally, `SoftmaxCrossEntropyForward` computes both the softmax probabilities and the cross entropy loss for each sample in the batch. The batch loss is copied back to the host and accumulated to compute the average epoch loss.

The backward pass reverses the order of operations. It begins with `SoftmaxCrossEntropyBackward`, which computes the gradient of the loss with respect to the fully connected output. Gradients for the fully connected weights and biases are computed using `FullyConnectedLayerBackward`, while `FullyConnectedBackward` propagates gradients back to the flattened representation. The gradient tensor is reshaped to the pooled format using `FlattenBackward`.

Next, `MaxPoolBackward` propagates gradients only to positions that were selected as maxima during the forward pass. It uses `AtomicAdd` to accumulate gradients mapping to the same input, avoiding race conditions. The gradient then passes through the ReLU activation of the second convolutional layer using `ReLUBackward`. Gradients for the second convolutional layer weights and biases are computed with `ConvLayerBackward`, and `ConvBackward` propagates gradients to the output of the first convolutional layer.

The same process is repeated for the first convolutional layer: ReLU backward is applied, followed by convolutional weight and bias gradient computation with ConvLayerBackward, and optional gradient computation with respect to the input images using ConvBackward.

After gradients are computed for all learnable parameters, the update step is performed. The SGD algorithm is applied separately to each set of weights and biases (Conv1, Conv2, and fully connected layer) using the `SGDBackward` kernel.

At the end of each epoch, the average loss is printed. Once all epochs are completed, CUDA events are used to measure and print the total training time.

The program then evaluates the trained model on the test set. Test images and labels are copied batch by batch to the GPU. The forward pass is executed exactly as during training, but without computing gradients. After the softmax step, predicted probabilities are copied back to the host. For each sample, the class with the highest probability is selected as the prediction. The number of correct predictions is counted to compute overall test accuracy. CUDA events are again used to measure testing time.

Finally, all allocated host and device memory is freed, including tensors for data, parameters, intermediate activations, and gradients. The program terminates after releasing all resources.

3.2. Optimized version

This implementation, in addition to the one mentioned above, introduces several optimizations that significantly improve computational efficiency and memory throughput compared to a naive CUDA implementation.

One major optimization is the use of `triple buffering` combined with CUDA streams. Three separate pinned host buffers and corresponding device buffers are allocated so that data transfers can overlap with kernel execution. While the GPU processes one batch, the CPU prepares and transfers the next one in parallel. Because pinned (page-locked) memory is used, transfers are faster and can truly run asynchronously. This overlap between communication and computation reduces idle GPU time and increases overall training throughput.

Another important optimization is `kernel fusion`, particularly in the convolutional layers where convolution and ReLU activation are combined into a single `convReluKernel`. By fusing operations, intermediate results do not need to be written to and read back from global memory between layers. This reduces global memory traffic, which is often the main bottleneck on GPUs, and improves cache utilization and latency.

The convolution kernels are designed to maximize parallelism across spatial positions, output channels, and batch elements. Grid dimensions are carefully structured so that each thread block works on a tile of the output feature map. In some forward configurations, `shared memory` is allocated dynamically to cache input tiles, reducing repeated global memory accesses. Since global memory access is significantly slower than shared memory, this tiling strategy improves memory bandwidth efficiency and arithmetic intensity.
The fully connected layer forward pass is implemented as a `tiled matrix` multiplication using shared memory. The input matrix and weight matrix are loaded in tiles into shared memory arrays, and partial products are accumulated before moving to the next tile. This reduces redundant global memory reads and leverages data reuse. Compared to a naive implementation where each thread reads directly from global memory for every multiply-add, this approach greatly improves performance and scales better with larger feature sizes.

Backward propagation kernels also contain optimizations. In particular, the convolution weight-gradient kernel uses `warp level primitives` __shfl_down_sync to perform reductions efficiently within a warp. This avoids slower atomic operations and reduces synchronization overhead. By performing partial sums inside warps and writing the result only once per parameter, contention and memory traffic are minimized.
The ReLU backward kernel uses the __ldg intrinsic for read-only data caching, which can improve memory access efficiency when reading activations during gradient computation. Similarly, the use of fmaf (fused multiply-add) in backward kernels increases numerical efficiency and leverages hardware instructions that compute multiplication and addition in a single step.
Finally, parameter updates are handled by a dedicated SGD kernel, allowing fully parallel in-place updates directly on device memory. This avoids unnecessary host-device synchronization and ensures that all gradient computations and weight updates remain on the GPU.

Overall, the main advantages of these optimizations are reduced global memory traffic, improved memory bandwidth utilization, better overlap between computation and data transfer and reduced synchronization overhead. Together, these choices transform a straightforward CUDA CNN implementation into a significantly more efficient and scalable training pipeline.

4. Experimental results

In the following sections, the experimental results are discussed in detail.

4.1. CPU/Python results

Results about the MNIST dataset and the FASHION dataset are reported in table (1) and in table (2) respectively.

Device	**Training Time [s]**	**Testing Time [s]**	**Test Accuracy**	**Pin memory**	**Num workers**
CPU	159.26	2.93	95.34%	-	4
CPU	184.99	3.04	95.34%	-	8*
CPU	155.17	2.93	95.34%	-	2
CUDA	19.12	0.66	95.34%	True	4
CUDA	19.55	0.72	95.34%	True	8*
CUDA	25.25	0.89	95.34%	True	2
CUDA	19.22	0.66	95.34%	False	4
CUDA	19.90	0.73	95.34%	False	8*
CUDA	24.97	0.87	95.34%	False	2

Table 1. Training and testing times and accuracy for the MNIST dataset for different devices and DataLoader settings. The "*" means that Pytorch library warns about the high number of $num\_workers$ and suggest to use 4. The "-" means that since CPU is used, `pin_memory` is ineffective. V100 GPU used.

Device	**Training Time [s]**	**Testing Time [s]**	**Test Accuracy**	**Pin memory**	**Num workers**
CPU	188.44	2.95	74.91%	-	4
CPU	181.07	3.98	74.91%	-	8*
CPU	189.96	3.02	74.91%	-	2
CUDA	19.15	0.66	74.91%	True	4
CUDA	19.37	0.73	74.91%	True	8*
CUDA	24.87	0.88	74.91%	True	2
CUDA	19.04	0.64	74.91%	False	4
CUDA	19.71	0.73	74.91%	False	8*
CUDA	24.84	0.86	74.91%	False	2

Table 2. Training and testing times and accuracy for the FASHION dataset using different devices and DataLoader settings. The "*" means that Pytorch library warns about the high number of num workers and suggest to use 4. The "-" means that since CPU is used, `pin_memory` is ineffective. V100 GPU used.

When we compare the results from the two tables across the MNIST and FASHION datasets, some clear patterns emerge that reflect both the characteristics of the datasets and the impact of execution configuration on training performance.

First, looking at accuracy, the MNIST dataset yields a high test accuracy of around 95.34% in all configurations, while the FASHION dataset achieves a significantly lower test accuracy around 74.91%. This difference is not caused by the training pipeline or hardware but by the inherent complexity of the tasks. MNIST consists of handwritten digits with relatively clear and distinct shapes, and even simple models can achieve near-perfect accuracy on it. In contrast, FASHION contains real world grayscale images of clothing items where subtle differences between classes make the classification task more challenging for a network of the same architecture and capacity. This results in lower baseline accuracy for FASHION when the same model and training setup are used, reflecting its higher complexity as a benchmark dataset. This behaviour happen also in the CUDA implementation.

Examining the training and testing times, we observe a dramatic performance gap between CPU and GPU executions on both datasets. GPU training (CUDA) completes the same number of epochs roughly seven to eight times faster than CPU training. This clearly highlights the benefits of parallel GPU computation for convolutional neural networks — even with simple models and small image datasets, GPUs can accelerate the large number of matrix operations involved in forward and backward passes. On MNIST, GPU training is typically around 19–25 seconds compared to 155–185 seconds on CPU; similarly for FASHION (19–25 seconds vs 181–190 seconds). This consistent almost $\times 10$ speedup in the best scenario (best training time vs worst training time) across datasets demonstrates that the advantage of GPU acceleration is largely independent of the dataset complexity itself and more about the architectural match between CNN computation and GPU parallelism.

Considering the DataLoader parameters, the number of workers ($num\_workers$) and whether memory is pinned (pin_memory), their impact is nuanced. Increasing $num\_workers$ beyond a moderate number provides diminishing returns or even regressions in performance. On the CPU, configurations with 8 workers often result in slower overall training compared to 4 workers, likely due to overhead and resource contention with too many parallel processes. On the GPU, changes in $num\_workers$ (2, 4, 8) do not significantly affect training time, indicating that the bottleneck is no longer data loading but GPU computation and data transfer. Moreover, unusually high values of $num\_workers$ can trigger warning messages from PyTorch that too many worker processes may slow down or freeze the data pipeline — because each worker consumes CPU and memory resources, the system can become overloaded rather than more efficient.

The effect of pin_memory is also visible: enabling it (True) results in marginally better GPU training times compared to False. Pinned memory on the CPU enables faster asynchronous transfers to the GPU, which helps keep the GPU fed with data and avoids idle periods while waiting for the next batch. However, its effect is more subtle in small image datasets like these because the overall transfer time per batch is small relative to the total computation time.

In summary, the comparative results show that the choice of dataset primarily influences classification accuracy — simpler datasets like MNIST yield high accuracy even with basic CNNs, whereas more realistic and structurally complex datasets like Fashion-MNIST present a harder challenge and lower accuracy under the same conditions. Meanwhile, the configuration of the training pipeline, especially using GPU acceleration, has a major impact on execution speed, and DataLoader parameters like $num\_workers$ and pin_memory have performance implications that are most noticeable when the GPU is heavily utilized.

4.2. Cuda results

In this section the results of the CUDA implementation are analyzed.

The results reported in tables (3), (5), (4) and (6) highlight two main aspects: the comparison between the naive and optimized implementations, and the performance differences across GPU architectures. Since the test accuracy remains constant across GPUs for each configuration, the observed differences are purely computational and not related to learning quality.

The most significant improvement comes from the optimized implementation. On the MNIST dataset, training time on the A40 decreases from 23.47 seconds in the naive version to 5.18 seconds in the optimized version, corresponding to a speedup of roughly $4.5\times$. A similar trend is observed on the RTX 6000 and the V100, where the optimized implementation consistently reduces training time by approximately four to five times. This demonstrates that the applied optimizations—such as better memory management, improved kernel design, and more efficient GPU utilization—have a substantial impact on performance.

A similar pattern is observed for the FASHION dataset. Training time decreases from about 23–25 seconds to roughly 5–6 seconds on the A40 and RTX 6000, and from over 41 seconds to about 8.5 seconds on the V100. The speedup remains consistent across datasets and hardware, indicating that the optimization strategy scales well and is not dataset dependent.

Regarding accuracy, small but consistent improvements are observed in the optimized version. On MNIST, accuracy increases from 97.26% to 98.08%, while on FASHION it increases from 87.18% to 87.53%. Since the network architecture is unchanged, this improvement likely results from better numerical stability given by not separating kernels.

When comparing GPUs, the A40 consistently achieves the fastest training times, followed by the RTX 6000, while the V100 is significantly slower. The same ranking holds for the optimized implementation. This behavior is coherent with architectural differences: newer GPUs such as the A40 benefit from more recent hardware optimizations and higher memory bandwidth.

Testing time is negligible compared to training time in all configurations. It remains around 0.09–0.13 seconds regardless of the implementation or GPU. Since testing involves only forward propagation, it is computationally lightweight relative to the backward pass and parameter updates required during training.

Overall, the optimized implementation delivers a substantial reduction in training time, approximately four to five times faster, while maintaining or slightly improving accuracy. The improvements are consistent across datasets and hardware platforms, demonstrating that the optimization strategy is effective, stable, and scalable.

The cumulative time per epoch Figure 1 clearly highlights the performance gap between CPU execution, a naive CUDA implementation, and the optimized CUDA version. The CPU configuration shows a linear increase reaching approximately 181 seconds after five epochs, confirming the significantly higher computational cost of training without GPU acceleration. The naive CUDA implementation drastically reduces training time, completing five epochs in about 25 seconds, demonstrating the advantage of parallel execution on GPU even without advanced optimizations. However, the optimized CUDA version further improves performance substantially, reducing total time to roughly 6.5 seconds after five epochs. This indicates a speedup of nearly 4× compared to the naive CUDA implementation and almost 28× compared to the CPU version. The curves labeled *CUDA pin True* and *CUDA pin False* overlap, suggesting that in this specific configuration pinned memory does not produce a measurable difference in cumulative epoch time, likely because data transfer is not the dominant bottleneck.

In Figure 2, the average loss curves show that all implementations converge correctly, but at slightly different rates. The CPU version starts with a significantly higher loss in the first epoch and decreases more gradually. Both

GPU	Training Time [s]	Testing Time [s]	Test Accuracy
A40	23.47	0.09	97.26%
RTX 6000	25.11	0.10	97.26%
V100	41.77	0.12	97.26%

Table 3. Training and testing times and accuracy for the MNIST dataset using the naive version

GPU	Training Time[s]	Testing Time[s]	Test Accuracy
A40	5.18	0.11	98.08%
RTX6000	6.43	0.13	98.08%
V100	8.50	0.12	98.08%

Table 4. Training and testing times and accuracy for the MNIST dataset using the optimized version

GPU	Training Time[s]	Testing Time[s]	Test Accuracy
A40	23.45	0.09	87.18%
RTX6000	25.09	0.10	87.18%
V100	41.65	0.12	87.18%

Table 5. Training and testing times and accuracy for the FASHION dataset using the naive version

GPU	Training Time[s]	Testing Time[s]	Test Accuracy
A40	5.17	0.11	87.53%
RTX6000	6.43	0.13	87.53%
V100	8.47	0.12	87.53%

Table 6. Training and testing times and accuracy for the FASHION dataset using the optimized version

CUDA implementations converge faster, with the optimized version achieving the lowest loss values at every epoch. By epoch five, the optimized CUDA implementation reaches the smallest average loss, indicating slightly more efficient training dynamics, possibly due to improved numerical efficiency and reduced overhead. Overall, the results demonstrate that while GPU acceleration already provides a major performance improvement, low level optimizations further enhance training efficiency without negatively affecting convergence behavior.

Regarding performance when changing the block size, Figures 3, 4, clearly show that block size has a significant impact on performance and numerical behavior in the naive CUDA implementation, while the optimized version remains stable across all tested configurations. From a GPU architecture perspective, this behavior is consistent with how thread blocks interact with Streaming Multiprocessors (SMs), register allocation, shared memory usage, and overall occupancy.

In the naive implementation, block sizes of 128 and 256 provide the best balance between parallelism and hardware resource usage. These configurations allow multiple thread blocks to reside concurrently on each SM, maintaining high occupancy and effective latency hiding. As a result, training time remains relatively low and convergence proceeds correctly.

When increasing the block size to 512, performance degrades. This likely occurs because larger blocks consume more registers and possibly more shared memory per block, reducing the number of active blocks that can be scheduled simultaneously on each SM. Lower occupancy limits the GPU's ability to hide memory latency, leading to longer execution times. Although the model still converges, both speed and final accuracy slightly deteriorate.

At a block size of 1024, the naive implementation fails entirely. Training time increases dramatically, and the loss remains around 2.303, corresponding to random guessing in a 10 class classification problem. This strongly suggests a resource saturation or synchronization issue. With 1024 threads per block, register pressure may become excessive, due to spilling register and reduced parallel efficiency. In extreme cases, incorrect indexing or insufficient synchronization can lead to invalid gradient updates, effectively preventing learning. From a GPU standpoint, this configuration pushes the kernel beyond a safe operational regime.

5. Conclusion

Three main conclusions can be drawn from the project. First, the largest improvement in absolute terms comes from

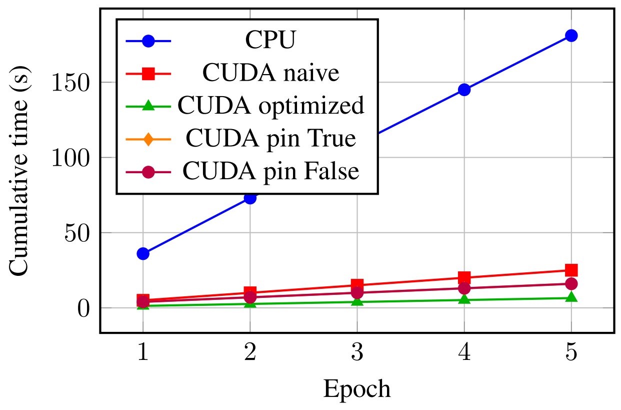

Figure 1. Cumulative time per epoch on RTX6000

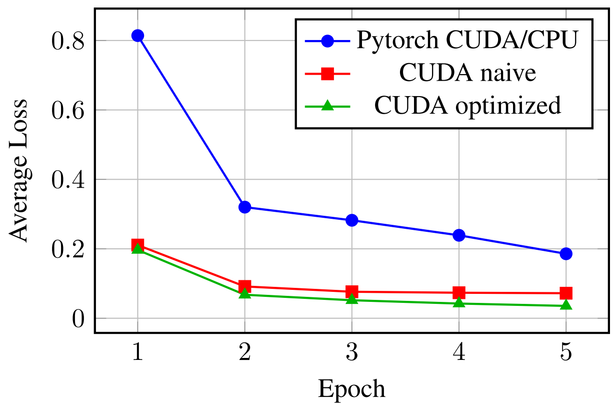

Figure 2. Average Loss value per epoch on RTX6000

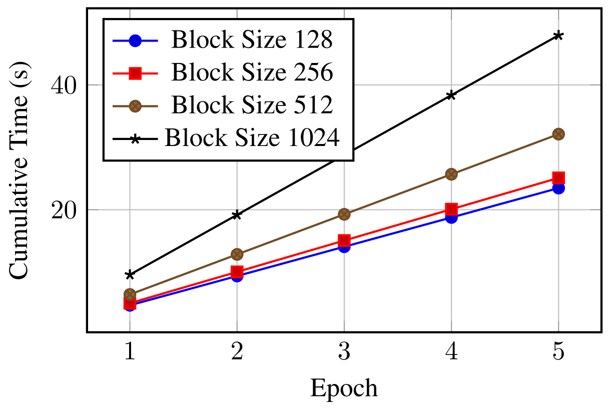

Figure 3. Cumulative training time per epoch for different block sizes, naive implementation on RTX6000

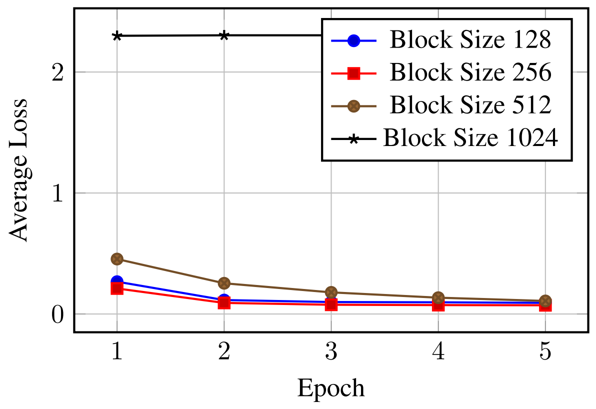

Figure 4. Average loss per epoch for different block sizes, naive implementation on RTX6000

moving computation from CPU to GPU (about 8× speedup). Second, once on GPU, algorithmic optimization yields another major improvement (around 4–5× speedup), which is larger than the differences caused by GPU generation alone. Third, DataLoader tuning (num_workers and pin_memory) has only a secondary impact compared to both hardware acceleration and kernel optimization.
Moreover, testing with different block sizes shows that optimal performance requires balancing thread level parallelism with hardware resource constraints and that increasing indefinitly the blocksize does not led a better performances by default.
Although the results obtained with the optimized from scratch implementation appear comparable to those achieved with PyTorch in terms of accuracy and convergence behavior, the underlying execution model is fundamentally different. PyTorch performs many additional operations in the background during training that are not explicitly visible to the user.
In contrast, the custom CUDA implementation is highly specialized: it computes only the strictly necessary operations. As a result, it can achieve lower execution times under controlled conditions. However, this performance gain comes at the cost of flexibility, maintainability, and ease of experimentation.

Regarding the overall accuracy, the CUDA implementation appears to achieve higher accuracy compared to the PyTorch version, this result should not be interpreted as a definitive indication of superior model quality. In deep learning, performance is highly sensitive to weight initialization, especially in relatively small networks trained on limited datasets. Even when reproducibility settings are enabled (e.g., fixed random seeds), small numerical differences between implementations can lead to different optimization trajectories. A particularly favorable initialization may guide the optimizer toward a better local minimum, while a less fortunate one may result in slightly lower accuracy. Therefore, the observed higher accuracy in the CUDA implementation should be considered a consequence of stochastic training dynamics and implementation specific numerical behavior, rather than intrinsic superiority over PyTorch.