David Rana(2246549) Dat H Huynh Khoi T Vo(1783957)

Team: Optimization

CNN implementation in CUDA

1. Introduction

This project aims to create a Convolutional Neural Network (CNN) model from scratch in C++, rather than using high-level Python libraries like TensorFlow or PyTorch, which already leverage GPU acceleration by default. The primary objective of this project is to implement the CNN using GPU parallel processing. We will develop custom CUDA kernels to perform CNN computations on the GPU efficiently and reliably.

2. Methodology:

2.1 CNN Development:

We have implemented CNN with one layer of a Convolution layer and one layer of a Pooling layer. Convolution layer implements six different filters that detect edge patterns and geometric structure after grayscaling the image. The convolution is performed by sliding each filter over the images, computing element-wise multiplication and sums at each position to produce six corresponding feature maps. These filters are a matrix of 3X3 that slides over the input image, computing the dot product between the filter and the image's region. This layer uses a stride of 1 and no padding, reducing the output size by 2 pixels in each dimension.

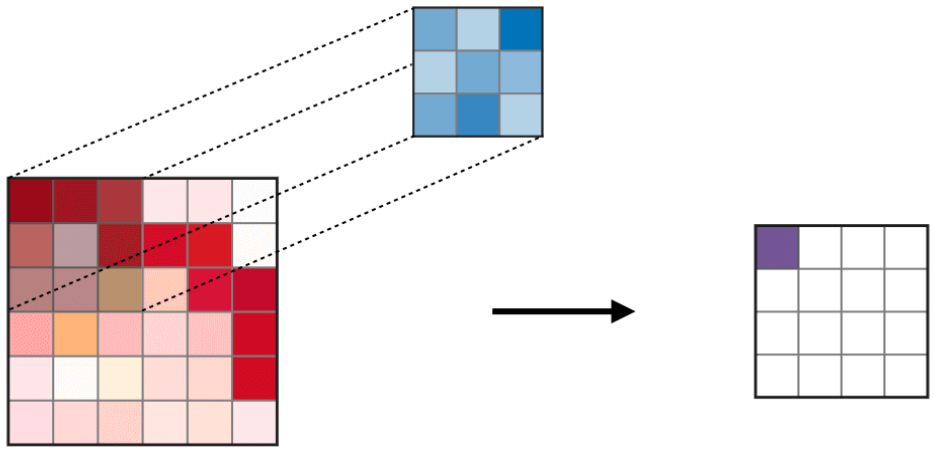

Figure 1. Convolution Layer

These generated images are passed to the downsampling layer of CNN, the Pooling layer. This layer divides the input images into non-overlapping regions and extracts the maximum value from each region. It shrinks the dimension of the image by the factor of the pool size. In our implementation, the pooling size is 3. So the resulting image is of the input dimension/ 3. We have also handled the cases when image dimensions aren't perfectly divisible by the pooling size.

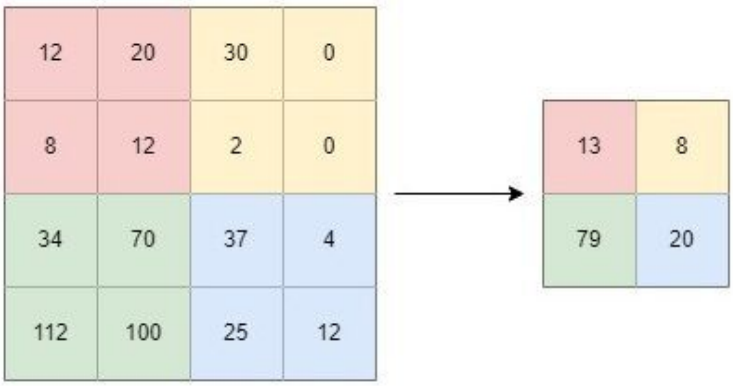

Figure 2. Pooling Layer

2.2 Project Structure:

```
├── CPU
│   ├── CNN_CPU.cpp
│   └── Makefile
├── Cuda
```

```
│   ├── Naive
│   │   ├── CNN_Naive_GPU.cu
│   │   └── Makefile
│   └── Optimize
│       ├── CNN_Optimize_GPU.cu
│       └── Makefile
├── Optimal_CPU
│   ├── CNN_CPU.cpp
│   └── Makefile
├── README.md
├── correctness.py
├── data
│   ├── images
├── dev
│   ├── CPU
│   │   ├── CNN_CPU.cpp
│   │   └── Makefile
│   └── GPU
│       ├── CNN_GPU.cu
│       └── Makefile
├── stb_image.h
```

Our project has different modules assigned for a specific purpose. The CPU module contains the final CPU implementation of the developed CNN. The GPU module contains the native and Optimal GPU implementation of the developed CNN. All the process images are inside the data folder -> images. We have another dev module that we used during the development of this project. In our final program, the program only produces the final images after processing. To make sure each filter and pooling is working during both implementations, we use the dev module program to save created images in each step to ensure our program is working as designed. Once we run our CPU/CNN_CPU.cpp and Cuda/*, they create subfolders in data cpu_output, gpu_naive_output, and gpu_optimal_output. All the images created by these programs are saved in those subfolders with their original name, with final_ in the beginning. We have Python scripts that check the correctness of our GPU implementation by comparing the

images produced in data/cpu_output with data/gpu_naive_output and data/gpu_optimal_output, which have the same filename. If there is any difference, then it will prompt the difference in the images, and if there is no difference, then our implementation is correct.

2.3 CPU Implementation:

We implemented the developed CNN layer on the CPU, which serves as the project's baseline and provides a correctness check for our GPU implementation. The CPU version processes images sequentially, applying six distinct 3×3 convolution filters (vertical, horizontal, two diagonal, X-shaped, and round patterns) to each grayscale image. Each convolution operation involves sliding the filter window across the image, computing element-wise multiplications and sums at every position, which reduces the output dimensions by 2 pixels in each direction. The resulting feature maps then undergo max-pooling with a 3×3 window, which downsamples each feature map by taking the maximum value in non-overlapping regions. The implementation handles edge cases where image dimensions aren't perfectly divisible by the pooling size. The program processes all images in the input directory, with an optional command-line argument to limit the number of images processed. This project utilizes OpenCV for image I/O and matrix operations. OpenCV's imread() function loads input images in grayscale format, converting them into single-channel matrices for efficient processing. The library's Mat class stores intermediate feature maps generated during convolution and pooling operations. For pixel-level computations, OpenCV provides direct access to matrix elements via at<uchar>(), enabling manual implementation of convolution filters and max-pooling. Finally, the processed images are saved using imwrite(), ensuring compatibility with standard image formats. While OpenCV offers optimized functions for many operations, this implementation manually handles convolution and pooling to maintain control over the CNN logic, with OpenCV serving as a reliable foundation for image handling and storage. It then prints the time program too to complete the CNN layer.

2.4 Native GPU Implementation:

The naive GPU implementation leverages CUDA to parallelize the CNN operations while maintaining the same six 3×3 convolution filters (vertical, horizontal, diagonal, X-shaped, and round patterns) as the CPU baseline. The implementation offloads both convolution and max-pooling operations to the GPU through dedicated kernels. For convolution, each thread

computes a single output pixel by performing the filter operation across a 3×3 neighborhood, with the round pattern filter stored in constant memory for faster access. The max-pooling kernel similarly parallelizes the downsampling operation by assigning each output pixel to a separate thread that finds the maximum value in its corresponding 3×3 region. The program handles memory transfers between host and device explicitly, allocating GPU memory for input images, intermediate feature maps, and final outputs. While maintaining the same functional behavior as the CPU version, this implementation processes multiple image regions simultaneously through CUDA's grid-stride approach, using 32×32 thread blocks for optimal GPU utilization. Edge cases are managed by conditionally executing threads only for valid output positions. The implementation includes comprehensive CUDA error checking and reports detailed GPU device information before processing. Like the CPU version, it processes all images in the input directory with optional command-line limits, using OpenCV for host-side image I/O operations while performing all compute-intensive operations on the GPU. It then prints the time program too to complete the CNN layer.

2.5 Optimize GPU Implementation:

The optimized GPU implementation combines convolution and pooling into a single fused kernel operation, significantly reducing memory transfers between operations. The implementation leverages CUDA's parallel architecture through several key optimizations:

1. A fused kernel performs both 3×3 convolution and 3×3 max-pooling in one pass, eliminating intermediate storage.
2. The convolution filter (X-shaped pattern) resides in constant memory for fast access.
3. Pinned (page-locked) host memory enables faster CPU-GPU transfers.
4. Multiple CUDA streams allow concurrent execution of memory transfers and kernel computations.
5. Loop unrolling pragmas optimize the convolution and pooling loops.
6. The restrict qualifier and __ldg() intrinsics improve memory access patterns.

The kernel uses a 32×8 thread block configuration optimized for modern GPU architectures, processing each output pixel in parallel. Boundary conditions are handled explicitly in the kernel to avoid memory violations. The host code implements a round-robin scheduling system across streams, overlapping memory operations with computation for different images. Performance metrics are tracked using CUDA events, providing accurate timing measurements. The implementation maintains the same filter patterns as the CPU version while achieving significantly higher throughput through these GPU-specific optimizations, processing standard

image formats (JPEG, PNG) via OpenCV's I/O functions while performing all compute operations on the GPU. It then prints the time program too to complete the CNN layer.

2.6 Correctness:

For the correctness check, we have created a Python script that compares the images that is created by the CPU CNN implementation with the GPU CNN implementation. It will flag a difference image with a greater than 0% difference between the images.

3. Implementation Challenges and Resolutions:

1. When designing the CNN layer, we faced uncertainty regarding how to implement the training process and verify correctness. Since training requires substantial computational resources and may yield inconsistent outputs for the same input, we decided to focus solely on implementing a single CNN layer rather than a complete CNN model. Our design intentionally omits activation functions and bias terms, as we are not training the model. Instead, we created one CNN layer specifically to compare its processed image data with our GPU implementation's output
2. Another challenge we encountered during development was working across different operating systems (Linux and Windows), which required us to implement cross-platform compatibility in our program.
3. One of the key challenges in implementing our CNN design was ensuring accurate data processing at each stage, particularly between the Convolution layer and the Max Pooling layer. Since the Max Pooling layer's output directly depends on the Convolution layer's results, we needed to verify proper execution at every step during development. To achieve this, we implemented intermediate image saving at each processing stage. This approach proved crucial when comparing final outputs between CPU and GPU implementations - if a correctness check failed, we needed to identify exactly where the calculations diverged. For this reason, we utilized the dev module to save intermediate images at each processing step, enabling us to systematically compare results between implementations and pinpoint any discrepancies.

4. Observation:

4.1 Number of images 100

- CPU Implementation

```
PS D:\cosc4397\optimization> ./CNN_CPU.exe
Current working directory: D:\cosc4397\optimization
Found 100 images
Time taken: 3884 ms
```

Figure 3. CPU Implementation of 100 images

- Naive GPU Implementation

```
PS D:\cosc4397\optimization> ./CNN_GPU.exe
CUDA Device Information:
----------------------
Number of CUDA devices: 1

Device 0: NVIDIA GeForce RTX 4090 Laptop GPU
  Compute capability: 8.9
  Total global memory: 16375 MB
  Multiprocessors: 76
  Max threads per block: 1024
  Max threads dimensions: (1024, 1024, 64)
----------------------
Input directory: .\data/training_set/100_images/
Output directory: .\data/gpu_naive_output
Starting processing of 100 images...

Processing complete!
================================
================================
Total images processed: 100
Failed to process: 0
Total processing time: 188 ms
================================
```

Figure 4. Naive GPU Implementation of 100 images

- Optimal GPU Implementation

```
PS D:\cosc4397\optimization> ./CNN_Optimal_GPU.exe
CUDA Device Information:
----------------------
Number of CUDA devices: 1

Device 0: NVIDIA GeForce RTX 4090 Laptop GPU
  Compute capability: 8.9
  Total global memory: 16375 MB
  Multiprocessors: 76
  Max threads per block: 1024
  Max threads dimensions: (1024, 1024, 64)
----------------------

Processing complete!
================================
Total images processed: 100
Failed to process: 0
Total processing time: 83 ms
================================
```

Figure 5. Optimal GPU Implementation of 100 images

4.2 Number of images 1000

- CPU Implementation

```
PS D:\cosc4397\optimization> ./CNN_CPU.exe
Current working directory: D:\cosc4397\optimization
Found 1000 images
Time taken: 41414 ms
```

Figure 6. CPU Implementation of 1000 images

- Naive GPU Implementation

```
PS D:\cosc4397\optimization> ./CNN_GPU.exe
CUDA Device Information:
----------------------
Number of CUDA devices: 1

Device 0: NVIDIA GeForce RTX 4090 Laptop GPU
  Compute capability: 8.9
  Total global memory: 16375 MB
  Multiprocessors: 76
  Max threads per block: 1024
  Max threads dimensions: (1024, 1024, 64)
----------------------
Input directory: .\data/training_set/1000_images/
Output directory: .\data/gpu_naive_output
Starting processing of 1000 images...

Processing complete!
================================
================================
Total images processed: 1000
Failed to process: 0
Total processing time: 1282 ms
================================
```

Figure 7. Naive GPU Implementation of 1000 images

- Optimal GPU Implementation

```
PS D:\cosc4397\optimization> ./CNN_Optimal_GPU.exe
CUDA Device Information:
----------------------
Number of CUDA devices: 1

Device 0: NVIDIA GeForce RTX 4090 Laptop GPU
  Compute capability: 8.9
  Total global memory: 16375 MB
  Multiprocessors: 76
  Max threads per block: 1024
  Max threads dimensions: (1024, 1024, 64)
----------------------

Processing complete!
================================
Total images processed: 1000
Failed to process: 0
Total processing time: 920 ms
================================
```

Figure 8. Optimal GPU Implementation of 1000 images

4.3 Number of images 8000

- CPU Implementation

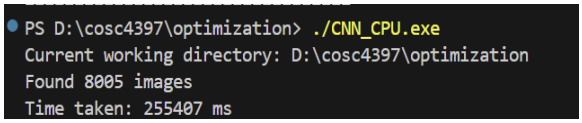

Figure 9. CPU Implementation of 8000 images

- Naive GPU Implementation

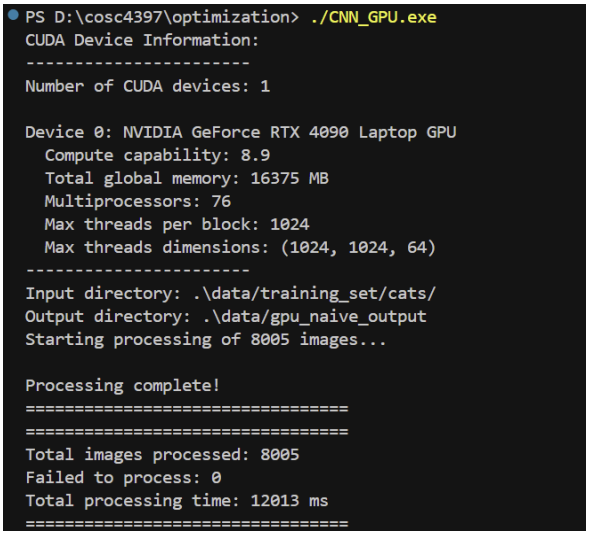

Figure 10. Naive GPU Implementation of 8000 images

- Correctness Test for Naive GPU Implementation

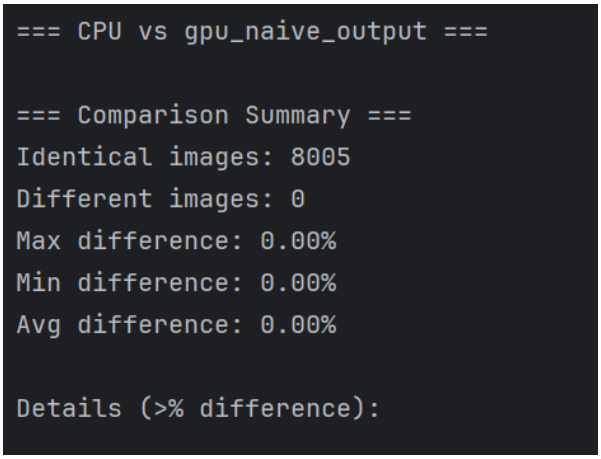

Figure 11. Correctness Test for Naive GPU Implementation of 8000 images

- Optimal GPU Implementation

```
PS D:\cosc4397\optimization> ./CNN_Optimal_GPU.exe
CUDA Device Information:
----------------------
Number of CUDA devices: 1

Device 0: NVIDIA GeForce RTX 4090 Laptop GPU
  Compute capability: 8.9
  Total global memory: 16375 MB
  Multiprocessors: 76
  Max threads per block: 1024
  Max threads dimensions: (1024, 1024, 64)
----------------------

Processing complete!
================================
Total images processed: 8005
Failed to process: 0
Total processing time: 7676 ms
================================
```

Figure 12. Optimal GPU Implementation of 8000 images

- Correctness Test for Naive GPU Implementation

```
=== CPU vs gpu_optimal_output ===

=== Comparison Summary ===
Identical images: 8005
Different images: 0
Max difference: 0.00%
Min difference: 0.00%
Avg difference: 0.00%

Details (>% difference):

Process finished with exit code 0
```

Figure 13. Correctness Test for Optimal GPU Implementation of 8000 images

4.4 Number of images 16000

- CPU Implementation

```
PS D:\cosc4397\optimization> ./CNN_CPU.exe
Current working directory: D:\cosc4397\optimization
Found 18000 images
Time taken: 758129 ms
```

Figure 14. CPU Implementation of 16000 images

- Naive GPU Implementation

```
PS D:\cosc4397\optimization> ./CNN_GPU.exe
CUDA Device Information:
-----------------------
Number of CUDA devices: 1

Device 0: NVIDIA GeForce RTX 4090 Laptop GPU
  Compute capability: 8.9
  Total global memory: 16375 MB
  Multiprocessors: 76
  Max threads per block: 1024
  Max threads dimensions: (1024, 1024, 64)
-----------------------
Input directory: .\data/training_set/1000_images/
Output directory: .\data/gpu_naive_output
Starting processing of 18000 images...

Processing complete!
==================================
==================================
Total images processed: 18000
Failed to process: 0
Total processing time: 27863 ms
==================================
```

Figure 15. Naive GPU Implementation of 16000 images

- Optimal GPU Implementation

```
PS D:\cosc4397\optimization> ./CNN_Optimal_GPU.exe
CUDA Device Information:
----------------------
Number of CUDA devices: 1

Device 0: NVIDIA GeForce RTX 4090 Laptop GPU
  Compute capability: 8.9
  Total global memory: 16375 MB
  Multiprocessors: 76
  Max threads per block: 1024
  Max threads dimensions: (1024, 1024, 64)
----------------------

Processing complete!
================================
Total images processed: 18000
Failed to process: 0
Total processing time: 17302 ms
================================
```

Figure 16. Optimal GPU Implementation of 16000 images

Final Result:

Number of Images	CPU (ms)	Naive GPU (ms)	Optimal GPU (ms)
100	3884	188	83
1000	41414	1282	920
8000	255407	12013	7676
18000	758129	27863	17302

Table 1. Performance Table

5. Conclusion:

The naive GPU implementation leverages CUDA to parallelize the CNN operations while maintaining the same six 3×3 convolution filters (vertical, horizontal, diagonal, X-shaped, and round patterns) as the CPU baseline. The implementation offloads both convolution and max-pooling operations to the GPU through dedicated kernels. For convolution, each thread computes a single output pixel by performing the filter operation across a 3×3 neighborhood, with the round pattern filter stored in constant memory for faster access. The max-pooling kernel similarly parallelizes the downsampling operation by assigning each output pixel to a separate thread that finds the maximum value in its corresponding 3×3 region. The program handles memory transfers between host and device explicitly, allocating GPU memory for input images, intermediate feature maps, and final outputs. While maintaining the same functional behavior as the CPU version, this implementation processes multiple image regions simultaneously through CUDA's grid-stride approach, using 32×32 thread blocks for optimal GPU utilization. Edge cases are managed by conditionally executing threads only for valid output positions. The implementation includes comprehensive CUDA error checking and reports detailed GPU device information before processing. Like the CPU version, it processes all images in the input directory with optional command-line limits, using OpenCV for host-side image I/O operations while performing all compute-intensive operations on the GPU. It then prints the time program too to complete the CNN layer.

6. Source Code:

Github: https://github.com/DavidRanaMagar/optimization

7. Contribution:

David Rana:

- Developing CNN model
- CPU implementation in linux
- Naive GPU implementation in linux
- Optimal GPU implementation in linux
- Correctness test
- Report
- Presentation

Khoi T Vo:

- Developing CNN model
- CPU implementation in windows
- Naive GPU implementation in windows
- Optimal GPU implementation in windows
- Report
- Presentation

Dat H Huynh:

8. Reference:

"Prompt." *ChatGPT*, OpenAI, [version if known], Date of interaction, https://chat.openai.com.

- For making makefile
- Debugging/Config libraries
- Details comments
- Suggestions on improving the kernels to achieve better performance

Github: https://github.com/warrenlyr/CUDA-CNN-From-Scratch

- This was the starting point of our project. We started out project with training the model, but that was not possible. In some research, we came across this repo and learn how they were implementing the CNN.