**Compilers and Programming Languages**
GPU Computing
WiSe 24/25

# Accelerating Convolutional Neural Networks through CUDA Optimizations on GPUs

René Richard Enjilian
Technische Universität Berlin

TECHNISCHE UNIVERSITÄT BERLIN

## Introduction

- **Goal**: Implement and optimize a CNN from scratch on a GPU using CUDA
- **Approach**: Created two CNN implementations on the GPU – a naive (unoptimized) baseline and an optimized using various CUDA techniques
- **Importance**: GPU acceleration reduces CNN training and inference times, making deep learning more efficient
- **Result**: Demonstrated notable performance improvements through targeted CUDA optimization techniques

## Convolutional Neural Network

- Convolutional Neural Networks (CNN) are neural networks specifically designed to handle structured data with spatial dependencies, such as images or videos.
- **Applications**: Widely used in image recognition, autonomous driving, medical imaging, and real-time video analysis

- **Convolutional Layer**
  - Apply kernels (filters) to input data to detect local spatial patterns:

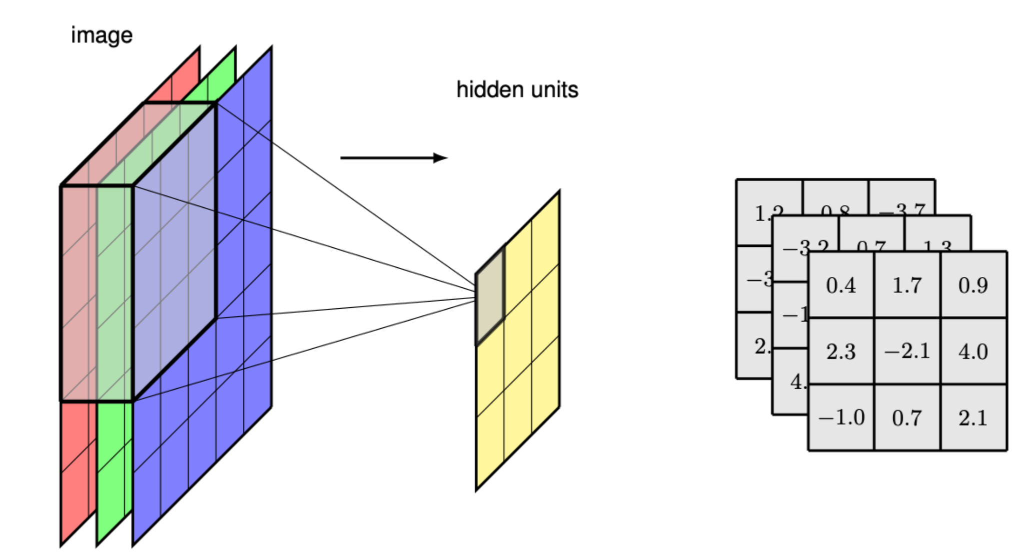


*Figure 1: Illustration of the convolution operation. An input image (left) is convolved with a kernel (right), producing a feature map (center).*

- **Rectified Linear Unit (ReLU)**
  - Non-linear activation function allowing the CNN to model complex, non-linear relationships in the data
  - $ReLU(x) = \max(0, x)$

- **Pooling Layer (Max-Pooling)**
  - Reduces spatial dimensions, decreases the number of parameters, and prevents overfitting:

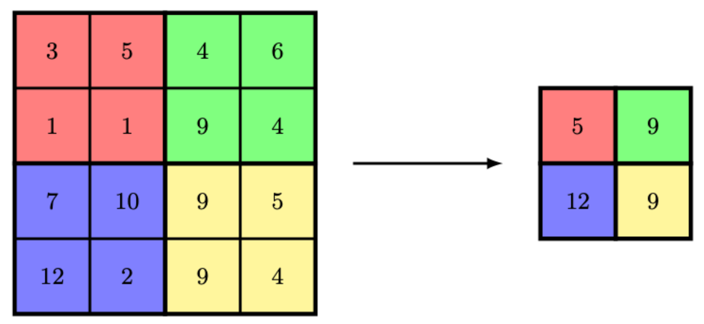


*Figure 2: Illustration of max-pooling, reducing each region (coloured) to its maximum value.*

- **Fully Connected Layer**
  - Combine extracted features to perform final classification
  - Each neuron connects to every output from the previous layer

- **Softmax Activation**
  - Converts raw scores into probabilities
  - Indicates the likelihood of each class
  - $Softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

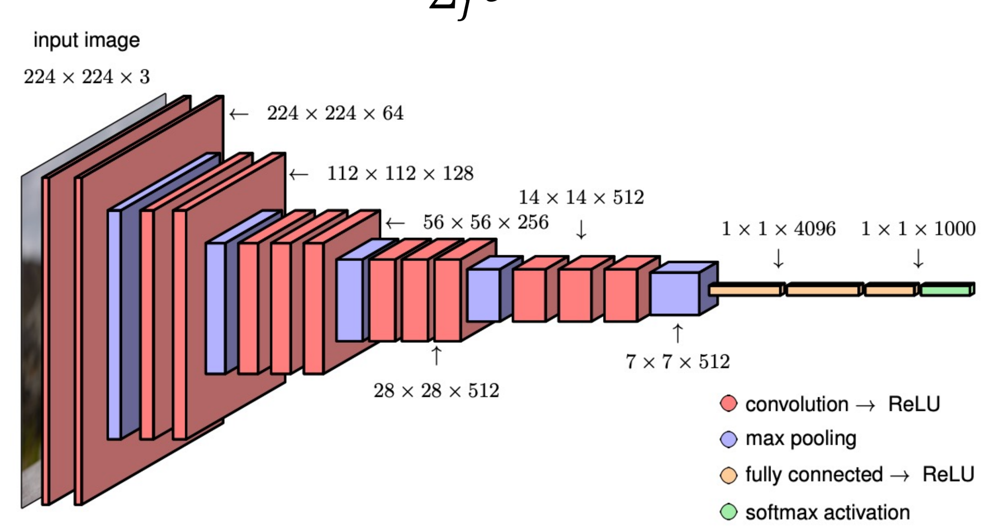


*Figure 3: Example CNN architecture (VGG-16).*

## Implementation & Optimizations

| Operation (Kernel) | Baseline (Naive) | Optimized | Performance Gains |
|---|---|---|---|
| Convolution + ReLU Forward | Separate kernels; Direct Global memory loads; No tiling | Kernel Fusion; Shared memory tiling; Constant memory (caching filters) | Fewer kernel launches; Less global mem I/O; Fast filter access |
| Max-Pooling + Flatten Forward | Separate kernels | Kernel Fusion | Fewer kernel launches; Less global mem I/O |
| Fully Connected Forward | Naive dot-product per neuron (loop-based) | GEMM-style matrix multiply with shared memory and tiling | Less global mem I/O (tiling); Better parallelization |
| Softmax + CrossEntropy Forward | Multiple loops for max, exp, sum; Repeated exponent calls | Single-pass approach with local arrays; Fast CUDA intrinsic __expf | Fewer repeated ops (exp, sum); Less overhead from looping |
| Fully Connected Backward | Two separate kernels: $\nabla W, \nabla b$ and $\nabla in$; Loops over batch per thread | Shared mem reduction for $\nabla W, \nabla b$; fmaf and loop unrolling for $\nabla in$ | Less global mem I/O; Parallel accumulation; Reduced loop overhead |
| Max-Pooling + Flatten Backward | Separate kernels; Uses atomicAdd | Kernel fusion; Direct index write (no atomic) | Fewer kernel launches; No atomic contention |
| Convolution Backward | Two separate kernels: $\nabla W, \nabla b$ and $\nabla in$; Triple nested loops | Warp-level reduction (__shfl_down_sync) for $\nabla W, \nabla b$; Per-sample shared mem for $\nabla in$ | Faster gradient reductions; Less global mem I/O |
| Data Transfer & Streams | Single CUDA stream | Multiple streams; Overlap H2D transfers with kernel execution | Higher GPU utilization; Less idle time on device |

## Evaluation

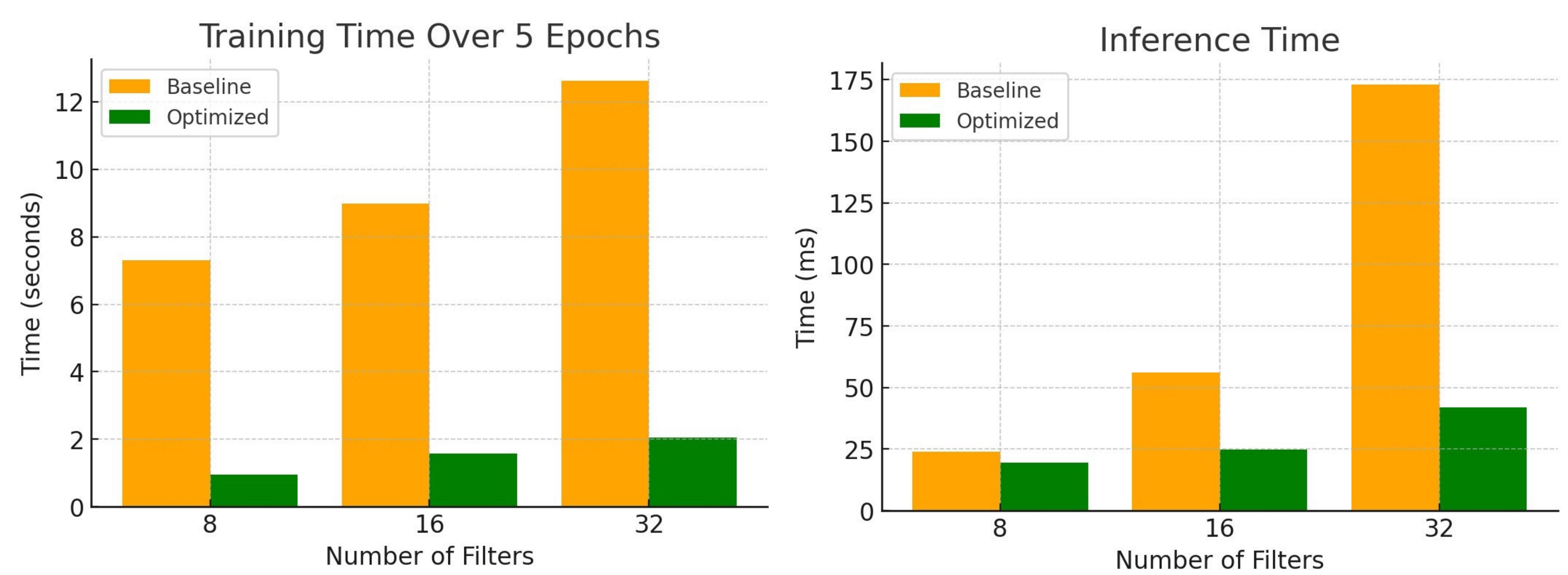


*Figure 4: Performance on MNIST dataset.*

- **Dataset**: MNIST
- **Batch size**: 64
- **Average Speedup**
  - Training: **6.5x**
  - Inference: **2.5x**

### References

**C.M. Bishop**, *Deep Learning: Foundations and Concepts*, Springer, 2024