Accelerating 2D Convolutions on GPUs

Vedant Bhasin, Vibha Arramreddy

Carnegie Mellon University

ABSTRACT

Background:

- Convolutions are extensively used in deep learning algorithms to learn feature dense representations across a variety of modalities such as computer vision, signal processing, and time series analysis.
- Convolutions have the benefits of being position invariant, and scaling well compared with dense layers, due to the filters being the only learnable parameters.
- Accelerating convolutions on GPUs can drastically reduce both inference and training latency on a variety of downstream tasks.

Project:

- We present various degrees of optimizations, profile and benchmark each iteration, and conduct extensive sensitivity studies and speedup analyses to give a comprehensive overview of the toolchains and frameworks available today for accelerated GPU kernels.

MOTIVATION & SCOPE

Goal 1: Writing a complete, generalizable high performance Conv 2d kernel that makes no assumptions on problem dimensions.

Motivation:

- Single high performance convolution kernel is hard but achievable.
- Scalable and generalizable high performance kernels is much harder

Goal 2: Compare and benchmark different degrees of hand tuned kernels with automatically generated high performance code.

Motivation:

- Writing hand-tuned GPU kernels is time-consuming and laborious
- Performance of hand-tuned GPU kernels doesn't necessary generalize across different hardware platforms.
- Surge in compiler toolchains that aim to provide hardware specific, high performance, automatic code generation.

DESIGN PRINCIPLES

Workload Distribution:

- Computation is sufficiently parallelized
- Even distribution across the CUDA Cooperative Thread Array (CTA).

Arithmetic Intensity:

- Decrease latency from memory IO as GPUs are memory bound
- Balance high arithmetic intensity with GPU utilization.

Memory Hierarchy:

- Use shared and register memory to boost performance.

Minimal Synchronization:

- Convolutions naturally have a high degree of parallelism
- Minimize thread synchronization in regards to shared memory access or reductions.

ALGORITHM

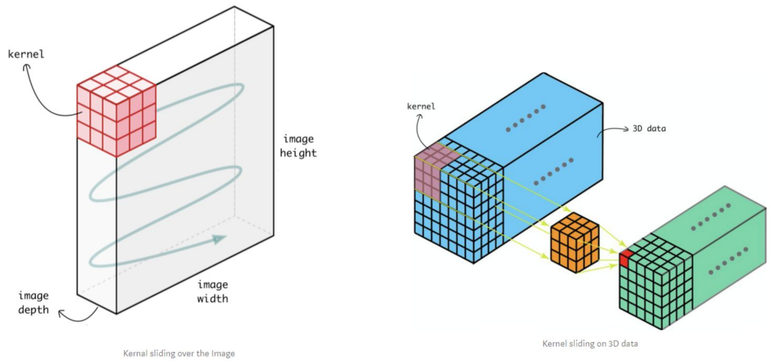

Figure 1: Visualization of a convolution typical of deep learning applications. The input is of shape Cin, H, W and the filter is of shape Cout, Cin, Kh, Kw. The key points to note here are that each output pixel is the summation over input channels of vector-vector dot products*

Algorithm Design:

- Block computes a partial result for a spatial region of the output. Specifically the contribution of the convolution of one input channel.
- Block loads input region into shared memory to increase arithmetic intensity
- Reduction across blocks is done through atomic adds to device memory.

Bottleneck:

- Atomic adds required to accumulate results in device memory.

Algorithm 1 Optimized 2D Convolution Kernel

Require: `d_result`: Output tensor in device memory
`d_x`: Input tensor in device memory
`s_x`: Input tensor in shared memory
`d_y`: Filter tensor in device memory
`Cin`, `H`, `W`, `Cout`, `K`: Input dimensions
Ensure: Compute the convolution result for each output pixel
Shared Memory Allocation: Allocate shared memory for input tiles of size $(\text{BLOCK\_DIM\_X} + K - 1) \times (\text{BLOCK\_DIM\_Y} + K - 1)$
Thread Mapping: Map thread indices to $x_{\text{out}}, y_{\text{out}}, c_{\text{in}}$
Iterate over output channels, c_{out}
Load Input Tile:
for all $(i, j) \in$ shared memory tile **do**
  Load corresponding elements from `d_x` to shared memory
end for
Synchronize threads
Compute Partial Convolution:
for $k_i = 0$ **to** $K - 1$ **do**
  for $k_j = 0$ **to** $K - 1$ **do**
    Accumulate (Atomic Add):

$$d_{\text{result}}[x_{\text{out}}, y_{\text{out}}, c_{\text{out}}] += s_{\text{x}}[x_{\text{out}} + k_i, y_{\text{out}} + k_j, c_{\text{in}}] \cdot d_{\text{y}}[k_i, k_j, c_{\text{in}}, c_{\text{out}}]$$

  end for
end for

Figure 2: Our optimized CUDA kernel, leveraging shared memory

PERFORMANCE

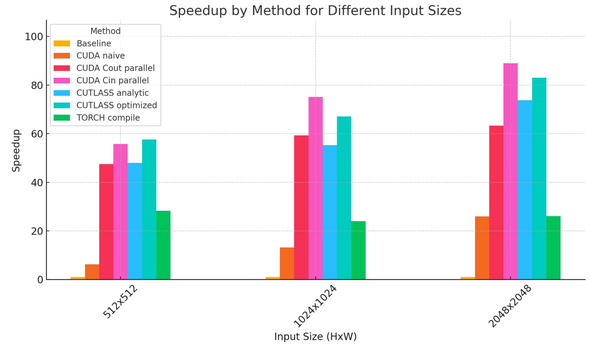

Figure 3: Sensitivity studies for input resolution.

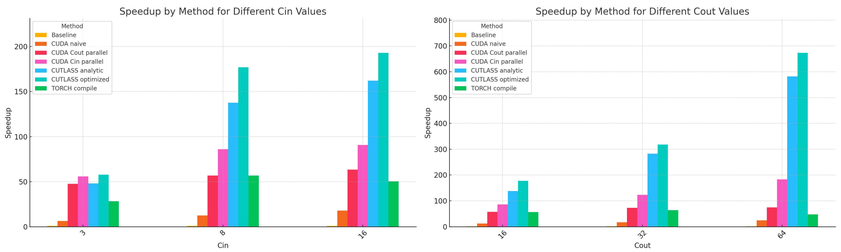

Figure 4: Sensitivity studies for input and output channels

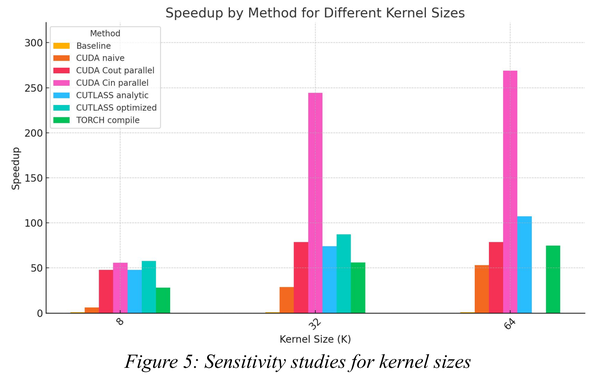

Figure 5: Sensitivity studies for kernel sizes

CONCLUSION

Optimized vs. Basic CUDA kernel:

- Significant performance improvements over baseline and naive methods
- Showcases strong scalability and efficiency across all tested parameters
- Manages increasing workloads effectively, as execution times for the optimized kernel grow much slower with larger input sizes, Cin, and Cout than baseline

Optimized CUDA vs. CUTLASS vs. Torch Compile:

- CUTLASS is the dominant method for large channels, significantly outperforming the optimized CUDA kernel in both speedup and execution times.
- CUTLASS has highly tuned implementations and efficient memory access strategies.
- Torch Compile lags behind both CUTLASS and the optimized CUDA kernel
- Torch Compile is a practical but less performant option
- Optimized CUDA kernel stands as a competitive, manually-tuned alternative

*https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610