ACCELERATING 2D CONVOLUTIONS ON GPU

Vedant Bhasin [*1] **Vibha Arramreddy** [*1]

ABSTRACT

We implemented an optimized 2D convolution kernel in CUDA, and benchmarked it against state-of-the-art CUTLASS kernels. Our approach shows two surprising findings 1. On GPUs improving arithmetic intensity even at the cost of workload distribution is highly beneficial. 2. Even simple optimizations such as shared memory usage can have significant boosts in performance. our approach achieves upto a **270x speedup** over our C++ baseline, and upto a **2.8x** speedup over a highly optimized CUTLASS kernel.

Project Page GitHub Repo

1 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. Therefore, accelerating convolutions on GPUs is of great importance since this can drastically reduce both inference and training latency on a variety of downstream tasks. In this project we present various degrees of optimizations, profile, and benchmark each iteration and conduct extensive sensitivity studies and speedup analyses giving a comprehensive overview of the toolchains and frameworks available today for accelerated GPU kernels.

1.1 Key Data Structures

The main data structures used in a 2D convolution include:

- **Input tensor**: A 3D tensor with dimensions C_{in} (input channels), H (height), and W (width).
- **Filter tensor**: A 4D tensor with dimensions C_{out} (output channels), C_{in}, K_h (kernel height), and K_w (kernel width).
- **Output tensor**: A 3D tensor with dimensions C_{out} (input channels), H' (height), and W' (width) where each element is the result of the convolution applied over the input tensor using the filter tensor, with dimensions determined by the size of the input, the filter, and the stride and padding used.

1.2 Key Operations

Convolution: The convolution operation calculates the output for each output channel as follows:

$$O[n,i,j] = \sum_{m=0}^{C_{in}-1} \sum_{p=0}^{K_h-1} \sum_{q=0}^{K_w-1} I[m, i+p, j+q] \times F[n,m,p,q]$$

where:

- $O[n,i,j]$ is the output value at output channel n and spatial position (i,j),
- $I[m, i+p, j+q]$ is the input value at input channel m and spatial position offset by (p,q),
- $F[n,m,p,q]$ is the filter value at output channel n, input channel m, and filter position (p,q),
- C_{in} represents the number of input channels,
- n indexes the output channels from [0, C_{out}).

At a fundamental level, for each output pixel we compute a vector-vector dot product for each C_{in} and then reduce across the C_{in} dimension. This algorithm can be parallelized across almost all dimensions with minimal dependencies.

1.3 Inputs and Outputs

- **Inputs**: The algorithm takes two primary inputs: the input tensor and the filter tensor.
- **Outputs**: The output is a tensor containing the convolved features, representing the combined effects of the filters applied to the input tensor.

1.4 Computationally Expensive Parts

The convolution operation itself is computationally expensive due to the nested loops required to perform element-wise multiplications and summations across large tensors. This is particularly intensive when dealing with high-dimensional inputs or large numbers of filters.

1.5 Workload Breakdown and Parallelism

- **Dependencies**: The main dependency in the convolution operation is the input data required for each kernel application. However, since different kernel applications do not depend on the results of others, they can be computed in parallel.
- **Parallelism**: There is significant data parallelism in convolution, as each output pixel's calculation is independent of others. This makes the algorithm highly amenable to parallel execution.
- **Locality**: Spatial locality is present in the access patterns of the input tensor, as neighboring input pixels are used together for multiple output calculations. Temporal locality is utilized in the repeated application of the same filters across different parts of the input.
- **SIMD Execution**: The element-wise multiplications and summations can be vectorized and efficiently executed using SIMD instructions, which are well-supported by modern GPUs.

2 APPROACH

2.1 Technologies Used

Our project was written entirely in CUDA, with Python bindings for easier testing. We target the Tesla T4 GPU. The machine specs are summarized in table 1. Notably, the Turing architecture supports Tensor Cores, which are utilized in the CUTLASS implementation. Another important aspect is that we have 48 KB of shared memory per block which means that we can load a maximum of 12,000 fp32 elements into shared memory at a time. Since our kernel is geared towards deep learning workflows, we make certain assumptions about the filter sizes to utilize shared memory, these are discussed further in section 2.2.

Property	Value
Name	Tesla T4
Compute Capability	7.5
Total Global Memory	14.75 GB
Shared Memory per Block	48.0 KB
Max Threads per Block	1024
Max Block Dimensions	1024 x 1024 x 64
Max Grid Dimensions	2147483647 x 65535 x 65535

Table 1. GPU Specifications for Tesla T4

2.2 Algorithm & Mapping

As mentioned earlier, partitioning the problem to maximize arithmetic intensity and utilizing shared memory were our key design principles. As such we make certain design decisions that increase arithmetic intensity and shared memory utilization at the cost of more synchronization. The algorithm and mapping is made explicit in algorithm 1. We structure our approach as follows:

Thread Block: Each block calculates the contribution of a particular input channel, for a spatial region, for a certain set of output channels. In other words, each block is mapped to a certain input channel, and output spatial region. The block then calculates the convolutions for a set of output channels, for the corresponding spatial region and in channel.

Thread: Each thread block within a thread is mapped spatially to one of the pixels in the region, and iterates over output channels for that pixel and input channel.

Algorithm 1 C_{in} Parallel 2D Convolution Kernel

Require: `d_result`: Output tensor in device memory
`d_x`: Input tensor in device 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 kernel receptive field 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 Receptive Field into Shared Memory:
for all $(x, y) \in$ threadBlock **do**
  for all $(i, j) \in$ receptiveField **do**
    $s_{\text{x}}[x+i, y+j] \leftarrow d_{\text{x}}[C_{in}, x_{out}, y_{out}]$
  end for
end for

Synchronize Threads

Compute Partial Convolution:
for all $c_{\text{out}} < Cout$, step size BLOCK_DIM_Z **do**
  for $k_i = 0$ **to** $K-1$ **do**
    for $k_j = 0$ **to** $K-1$ **do**

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

    end for
  end for
end for

Pros: Leverages spatial and temporal locality. The shared input tile is accessed for multiple pixels across multiple tiles, and the mapping of the input onto the CUDA grid exploits the spatial locality present in the problem.

Cons: Repeated atomicAdd operations to device memory are bad for performance because they serialize threads, forcing them to update the memory location one at a time, reducing parallelism. This creates contention, especially when many threads access the same memory location, leading to delays. Furhtermore, repeated accesses to global memory incur high latency penalties.

2.3 Previous Iterations

Before arriving at our final algorithm, we tried many different solutions that explored different trade-offs between work distribution, arithmetic intensity, and synchronization costs. One other algorithm we implemented was an approach that parallelized over height, width, and output channels.

Out-channel-parallelized approach: In this approach, each block was mapped to a certain output channel and output spatial region. The block calculates the convolutions for a set of input channels for its corresponding spatial region and output channel. It does this by assigning each thread in the block to one of the pixels in the spatial region. Each thread iterates over the input channels for its assigned pixel and the block's output channel.

Pros: This approach eliminates the main bottleneck in our final algorithm: the atomicAdd operation. Since each block is assigned an output channel, the synchronization happens within a block. Therefore, blocks are no longer written to the same locations in device memory, and atomicAdd operation is no longer necessary.

Cons: In order to minimize the latency of memory operations and increase arithmetic intensity, we loaded portions of the input tensor into shared memory. This is similar to the approach we took in our final algorithm. However, when iterating over input channels, the portion of the input tensor that each block covers is much larger than when iterating over output channels. Therefore, we used a tiling approach to periodically reload portions of the input tensor based on the input channel. However, reloading the input tensor was very time consuming.

Comparison: Although this approach eliminated the bottleneck of the atomic adds, it required a new, more costly operation. Having to reload the input tensor ended up being more time-consuming than the synchronization cost of the atomicAdd operations. We speculate that this is due to the fact that, since we set the block size to maximum possible size for the GPU and optimize for a small number of input channels, the atomicAdd operations do not face too much contention.

Algorithm 2 C_{out} Parallel 2D Convolution Kernel

Require: `d_result`: Output tensor in device memory
`d_x`: Input tensor in device 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 kernel receptive field of size $(\text{BLOCK\_DIM\_X}+K-1)\times(\text{BLOCK\_DIM\_Y}+K-1)$

Thread Mapping:
Map thread indices to x_{out}, y_{out}, and c_{out}

Load Receptive Field into Shared Memory:
for all c_{in} **in** $[0, Cin)$, step size BLOCK_DIM_Z **do**
  for all $(x, y) \in$ threadBlock **do**
    for all $(i, j) \in$ receptiveField **do**
      $s_{\text{x}}[x+i, y+j] \leftarrow d_{\text{x}}[C_{in}, x_{out}, y_{out}]$
    end for
  end for

Synchronize Threads

Compute Partial Convolution:
Initialize $local\_sum \leftarrow 0$
for $k_i = 0$ **to** $K-1$ **do**
  for $k_j = 0$ **to** $K-1$ **do**
    Compute:

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

  end for
end for

Update result:

$$d_{result}[c_{\text{out}}, x_{\text{out}}, y_{\text{out}}] \mathrel{+}= local\_sum$$

Synchronize Threads
end for

3 RESULTS

3.1 Baselines

We evaluate seven different Conv 2D variants to benchmark our implementations:

Baseline: A C++ implementation with nested loops compiled with `O4`.

CUDA naive: A naive CUDA implementation with the same mapping as our C_{in} parallel approach but no shared memory usage.

CUDA C_{out} Parallel: Our C_{out} parallel CUDA implementation described in section 2.3.

CUDA C_{in} Parallel: Our C_{in} parallel CUDA implementation described in section 2.2.

CUTLASS Analytic: An optimized CUTLASS implementation that utilizes tensor cores and thread, warp, and block level tiling. Makes no assumptions on input sizes and layout.

CUTLASS Optimized: A highly optimized CUTLASS implementation that relies on input alignment and kernel dimensions being smaller than 32.

TORCH Compile: An advantaged compilation-based approach to generating hardware-specific optimized code based on the MLIR toolchain.

3.2 Speedup across Input Size

Speedup graphs across input sizes are displayed in fig. 1. Our C_{in} parallel method shows strong performance even outperforming the CUTLASS optimized implementation. Both our C_{in} and C_{out} parallel methods scaled well with spatial dimensions. This is due to the fact that both our approaches are highly parallelized across spatial dimensions utilizing shared memory to exploit spatial locality.

3.3 Speedup across Number of Input Channels

Speedup graphs across number of input channels are displayed in fig. 2. Our C_{in} parallel method uses `atomicAdd` to reduce across the input channel dimension, which scales poorly. This is because as the number of input channels increases, the number of threads within different block attempting to write to the same location in device memory increases. Therefore, there is more contention for the `atomicAdd` operations. C_{out} parallel iterates over the input channels and repeatedly loads tiles to shared memory, which also scales poorly with respect to input channels due to an increased amount of repeated loads.

3.4 Speedup across Number of Output Channels

Speedup graphs across number of input channels are displayed in fig. 3. Here, our methods are dwarfed by the CUTLASS baselines. C_{in} parallel scales poorly with the number of out channels since it linearly increases the number of `atomicAdd`s per block, which increases contention significantly. For C_{out} parallel, increasing the number of out channels linearly scales the number of shared memory loads, since more blocks load the same regions of device memory into shared memory.

3.5 Speedup across Kernel Size

Speedup graphs across kernel sizes are illustrated in fig. 4. Surprisingly, our C_{out} parallel approach significantly outperforms the strong CUTLASS baselines, even without Tensor Core utilization. This is likely due to the extremely high arithmetic intensity achieved by loading an input tile into shared memory, and convolving it with all output kernels. With large kernel sizes, this would have significant latency as the number of device memory reads for the input values would scale poorly.

4 CONCLUSION & FUTURE WORK

Our evaluation demonstrates that both the C_{in} and C_{out} parallel methods show strong performance across various input configurations. The C_{in} parallel approach outperforms even the optimized CUTLASS implementation for large spatial dimensions and kernel sizes, leveraging efficient shared memory utilization. However, both methods fall short when scaling across input and output channels due to the contention introduced by repeated `atomicAdd` operations and excessive shared memory loads.

For future work, we aim to address these limitations by:

Reducing reliance on `atomicAdd` operations through further tiling or partitioning across the channel dimensions. Incorporating Tensor Core acceleration via the `CUDA WMMA API` to enhance performance. Designing optimized kernels for small kernel sizes to better utilize register memory, minimizing shared memory overhead. These improvements will help bridge the gap in performance across channel dimensions while maintaining the strengths of our current implementations.

*Equal contribution [1]Carnegie Mellon University, Pittsburgh, PA, USA. Correspondence to: Vedant Bhasin <vedantbhasin@cmu.edu>, Vibha Arramreddy <varramre@andrew.cmu.edu)>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2024.

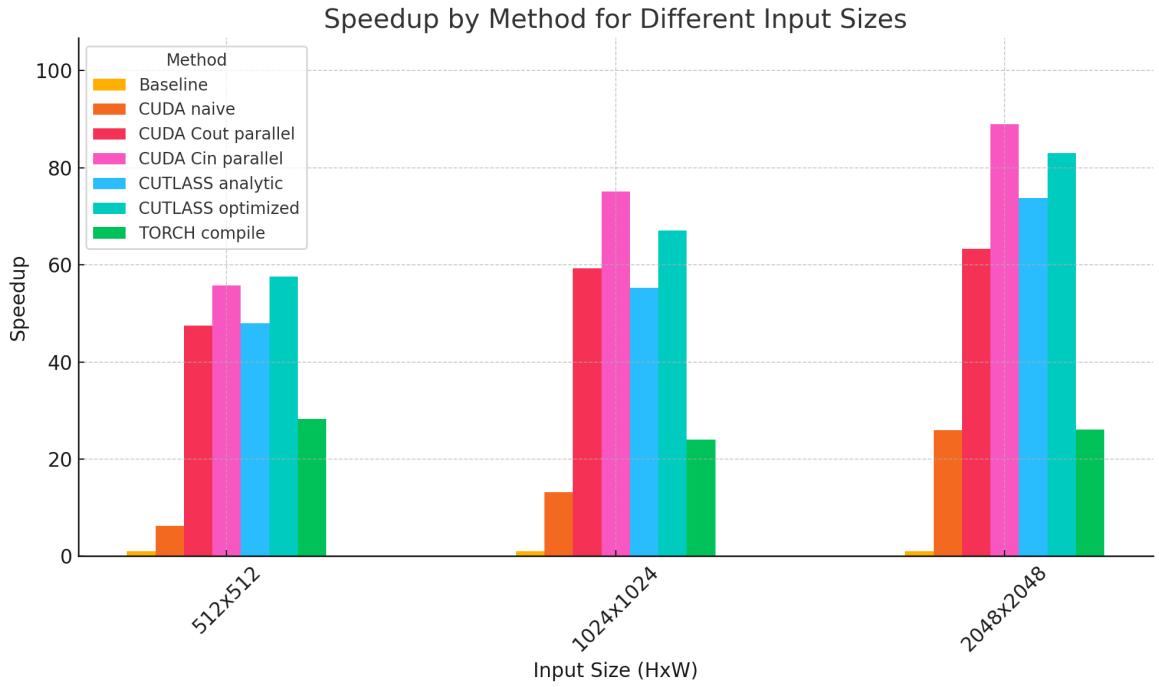

Figure 1. This figure shows speedup for different methods with increasing spatial dimensions. Our C_{in} parallel method shows strong performance even outperforming the CUTLASS optimized implementation. Both our C_{in} and C_{out} parallel methods scaled well with spatial dimensions.

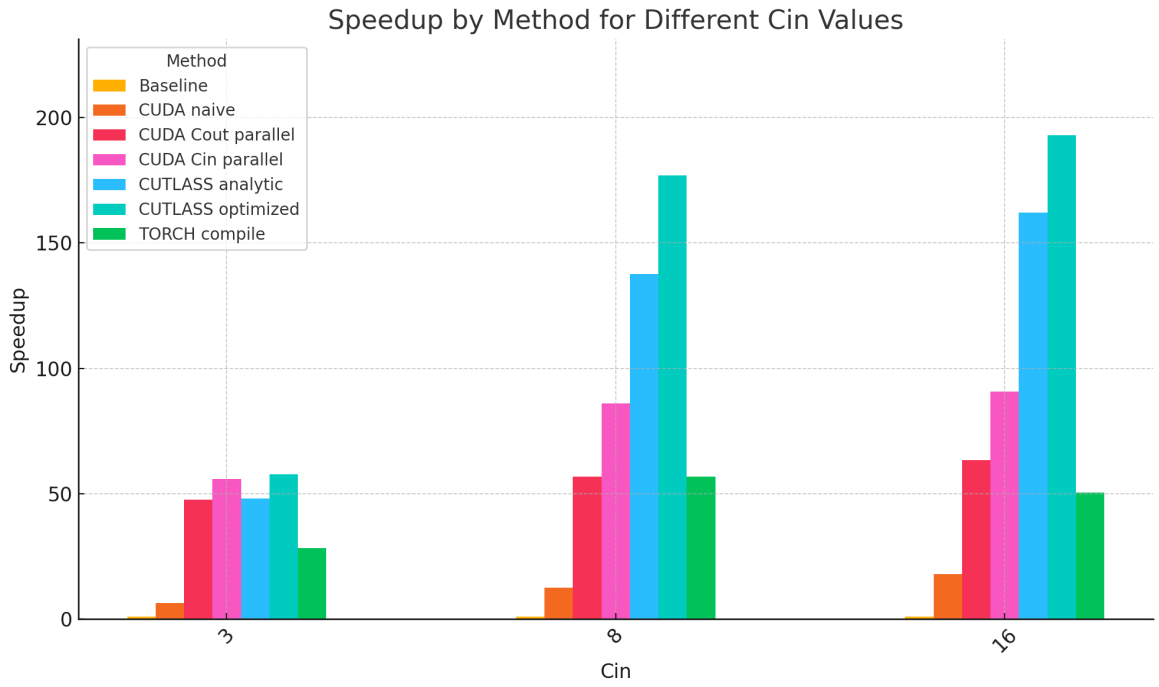

Figure 2. This figure shows speedup for different methods with increasing C_{in} dimensions. Here, our methods are outperformed by the CUTLASS baselines. Our C_{in} parallel method uses `atomicAdd` to reduce across the input channel dimension which scales poorly. C_{out} parallel iterates over input channels and repeatedly loads tiles to shared memory which also scales poorly with input channels due to the repeated loads.

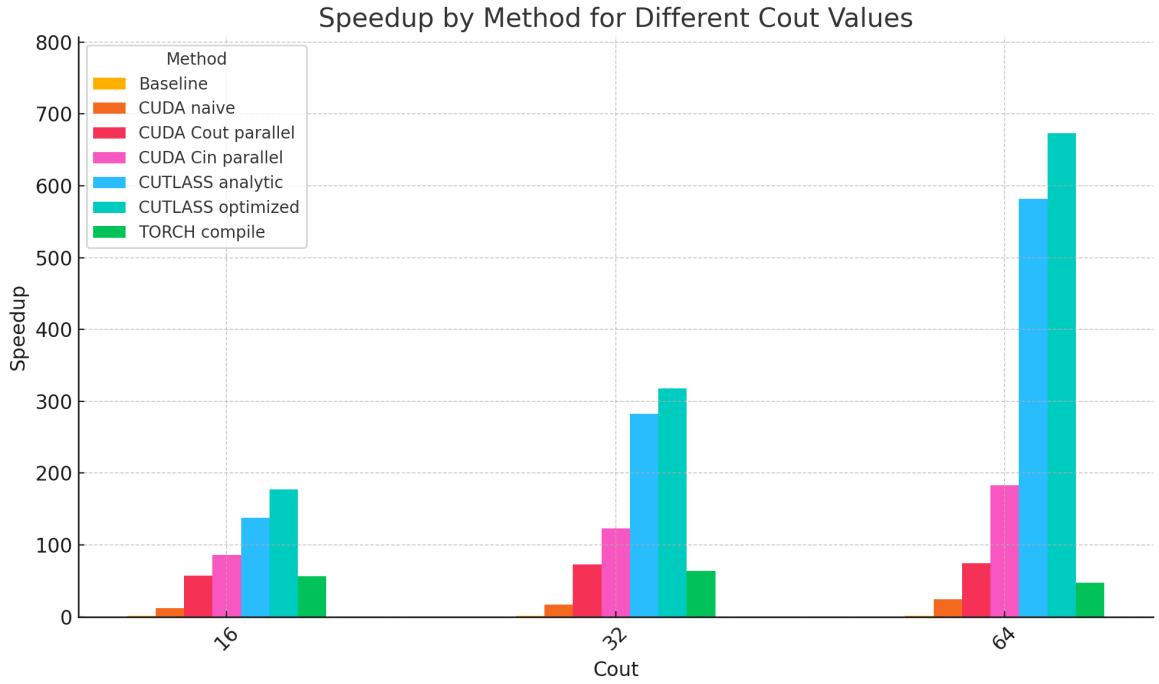

Figure 3. This figure shows speedup for different methods with increasing C_{out} dimensions. Here, our methods are dwarfed by the CUTLASS baselines. C_{in} parallel scales poorly with the number of out channels since it linearly increases the number of `atomicAdd`s per block which increases contention significantly. For C_{out} parallel, increasing out channels linearly scales the number of shared memory loads, since we more blocks loading the same regions of device memory into shared memory.

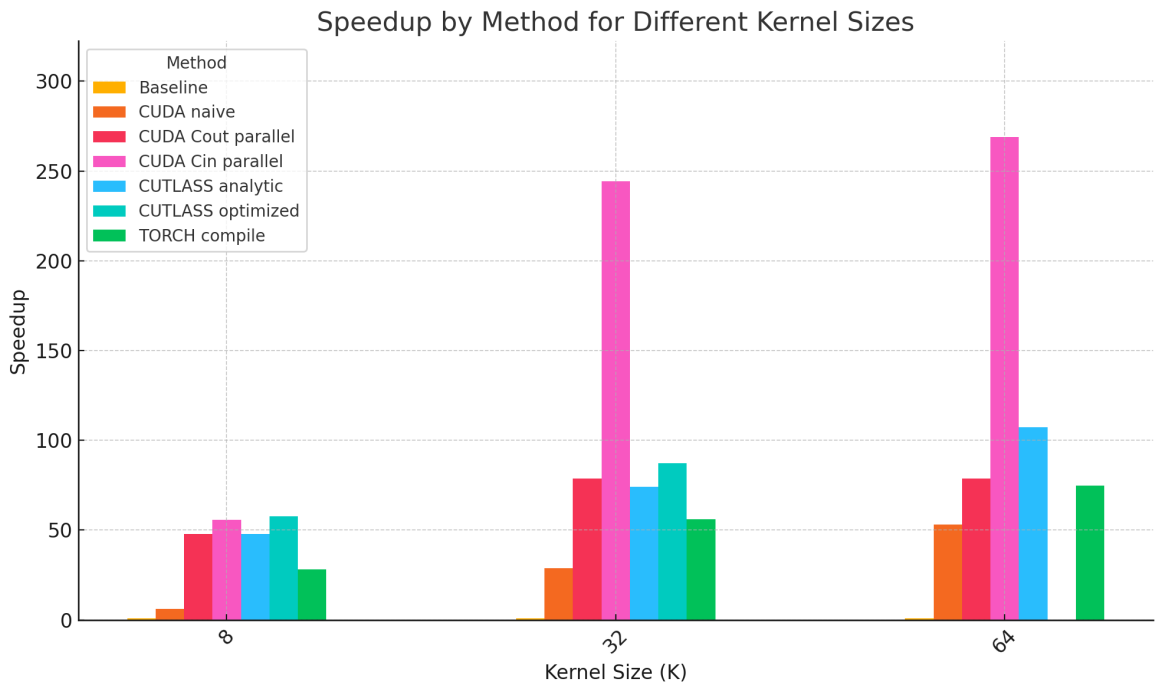

Figure 4. This figure shows speedup for different methods with increasing K dimensions. Our C_{in} parallel outperforms all other methods including the optimized CUTLASS kernel. This is likely due to the fact that, increasing kernel size leads to better shared memory utilization with a constant input size. C_{out} parallel still suffers from the reloading issue which decreases shared memory utilization even as the kernel size increases. **N.B.** for kernel size 64 we don't have data for CUTLASS optimized since this configuration requires $K \leq 32$