CS 267 Final Project: Communication-Avoiding Depthwise-Separable Convolutions

Group 54: Connor Lien, Saurav Banka, Tejvir Jogani

Spring 2022

1 Introduction

Convolutions are ubiquitous in deep learning applications such as facial recognition, object detection, and image analysis. Recently, depthwise-separable convolutions have increased in popularity in models such as MobileNets and SOTA models such as ConvNeXt since they require fewer multiplications than ordinary convolutions. Our goal was to optimize direct depthwise-separable convolutions for communication to attain better performance than current convolution implementations.

Typically, convolutions are implemented by reduction to matrix multiply using im2col since the former is highly optimized. However, we can achieve a lower communication bound by blocking direct convolutions, and attain additional speedups due to the operation's embarrassingly parallel nature. We compute depthwise-separable convolutions directly with communication-avoiding tilings proposed by Dinh et al., and other optimizations such as loop reordering and a microkernel[1]. We parallelize our serial code using OpenMP for CPU and CUDA for GPU data-level parallelism. Finally, we develop a custom PyTorch extension to benchmark our implementation as a drop-in convolution layer in existing neural networks.

2 Depthwise Separable Convolutions

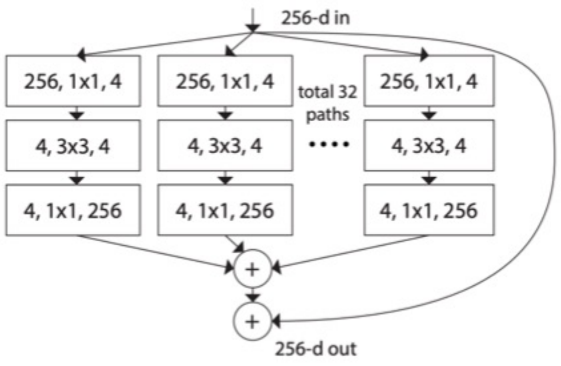
Typical convolution operations are generally performed using the Conv2d operation on an input. However, with the rise of new SOTA models like MobileNet and ConvNext, depthwise separable (DWS) convolutions have gained a lot more popularity due to speed and model performance improvements.

In depth wise convolutions, the filter and input are split by channels, convolved separately, then concatenated. A pointwise convolution is then applied. We can increase the number of channels and maintain roughly the same computational complexity with this technique, while

[1]Our code can be found at www.github.com/connorlien/cs267-project

improving model performance. This also results in significantly fewer operations and hence faster performance.

Figure 1: DWS Convolution Architecture



As visible in the above diagram, an input of d channels is split up and convolved separately by d filters. This naturally yields itself to a parallel friendly approach as described in our report. [4]

3 Parallelism

3.1 Instruction Level Parallelism with SIMD Microkernel

In order to speed up performance, we implement two SIMD Microkernels for depthwise and pointwise convolutions respectively. SIMD Instructions are vector instructions that operate on multiple data view SIMD registers. We use AVX512 intrinsics, which allow us to operate on 16 single precision numbers at a time.

Our implementation of the depthwise convolution microkernel uses three AVX512 registers (type `_mm512`). The first one holds all the channels for image I at batch B, width W, and height H. The second one holds the filter F in a similar way and third holds the output O in a similar way. The following algorithm describes how we are able to compute $O[B][c][W][H]$ $\forall c \in C$ where C is the number of output channels.

Algorithm 1 SIMD Microkernel for Depthwise Seperable Convolution Operation

```
Load Filter $f$ into Register B
Load Output $o$ into Register C
**for** $h_f$ in $0...H_f$ **do**
    **for** $w_f$ in $0...W_f$ **do**
        Load Input $im$ into Register A
        FuseMultiplyAdd($im, f[h_f, w_f], O$)
        Store $o$ back in $O$
```

Another quick optimization we made for this kernel was loading the filters into SIMD registers once and pass in the registers as arguments into the microkernel. This is since the filter was the same across the input so we we able to get a 2x performance boost by simply reusing the registers and being more communication efficient.

Our implementation of the pointwise convolution microkernel involved using the `__mm512_reduce_add_ps` intrinsic to sum up the produce of the filter and image across channels (essentially computing a weighted average). The following algorithm better describes this.

Algorithm 2 SIMD Microkernel for Pointwise Convolution Operation

```
Load Input $im$ into Register $A$
Load Filter $f$ into Register $B$
Register $C$ = Multiply($A$, $B$)
$O+$ = ReduceAdd($C$) =0
```

We tried using a copy optimization to copy over data to `__mmalloc`'d memory. However, as our input tensor sizes scale up, copy optimization costs dominate leading to poor performance. Hence we use unaligned load and store intrinsics.

3.2 Process Level Parallelism using OpenMP

A convenient way to improve the performance of our code was to run it in parallel over multiple cores. Our approach used OpenMP as both our depthwise and pointwise convolutions are embarrassingly parallel.

Our implementation uses a single `#pragma omp parallel for` for both convolution operations. We also used the `collapse(4)` setting for the depthwise convolution and `collapse(3)` setting for the pointwise convolution, as we found parallelizing over multiple loops together produced the best performance.

One hazard we worked to avoid were race conditions. To ensure this, we decided to parallelize over independent feature dimensions. The depth-wise feature maps are independent across the channel dimension. Additionally, each output pixel depends on exactly one patch of pixels in the input tensor. Hence we can collapse all loops except for loops across the filter tensor's height and width filters. Similarly the output feature maps of pointwise convolutions have dependencies only across the channel dimension. Hence all loops except for the channel dimension can be collapsed using OpenMP. Alternatively, we tried using C++ atomics. However, there was a lot of overhead in converting existing single precision tensors to atomic long tensors, which led to a slowdown. Atomic add could potentially be used to update tensor values.

3.3 GPU Accelerated CUDA Kernel

Similar to OpenMP, we easily parallelize over our convolution code using CUDA. CUDA has built-in support for up to three dimensions of threads: x, y, and z. This allows us to remove three loops from our code, as each thread is assigned to a specific combination of the three variables we assign to each dimension of threads. Threads with variables that are out-of-bounds return early so they do not run.

We parallelize over batch, output tensor width, and output tensor height respectively since those variables are the largest and result in maximum GPU usage. To avoid conflicting writes, we specifically assign each thread to compute a location of the output instead of process a portion of input. We test several thread counts per block and find that 32 threads per block leads to the best performance.

In addition, we implement blocking for our GPU version in an attempt to improve performance. However, we find that this only adds additional overhead. This is likely because our current memory accesses in our kernel are already fairly close together, and do not result in many GPU cache misses.

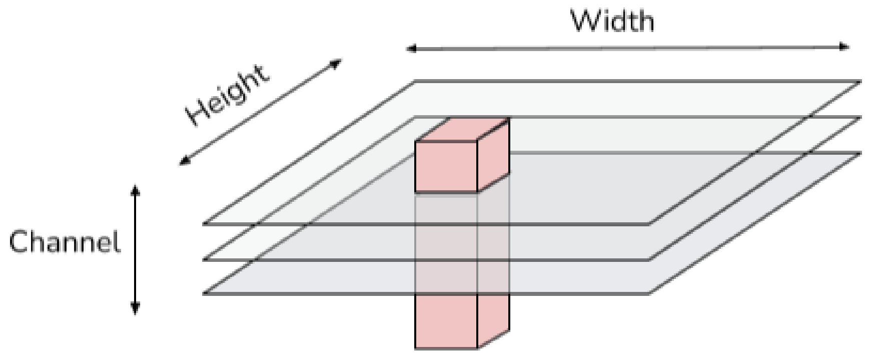


Figure 2: Depthwise convolution computation on GPU. Thread (x, y, z) computes filter multiplications channel-wise along the input tensor for indexes (batch, width, height)

4 Other Optimizations

4.1 Data Storage

We store our data in a naive format for convolutions to match the data storage format of PyTorch. The outer dimension is batch, followed by channel, height, and width respectively. This reduces our overhead when creating the PyTorch extension, since the input and output tensors can be directly created from a blob in memory. Additionally, row-major storage along the height and width dimensions leads to easier blocking along the width and height dimensions of the large input and output

4.2 Loop Ordering

Modifying the loop ordering for optimal cache access patterns was the most significant improvement we made to our serial code, speeding up our execution time by 13x for larger tensor sizes.

Since inner loops are executed a higher number of times, we can minimize cache misses by structuring the loops to follow the data storage format (explained above). This means the most commonly changed variable is width, which has the highest chance of staying within a cache line as elements are contiguous along the width-dimension of the tensor. Batch is our outermost loop, since changing batch frequently will lead to accesses in different cache lines and incur a performance penalty.

Algorithm 3 Optimized Loop Ordering for Depthwise Convolution

```
for $b$ in 0...$B$ do
    for $c$ in 0...$C_{in}$ do
        for $h$ in 0...$H$ do
            for $w$ in 0...$W$ do
                for $h_f$ in 0...$H_f$ do
                    for $w_f$ in 0...$W_f$ do
                        Convolve
```

Algorithm 4 Optimized Loop Ordering for Pointwise Convolution

```
for $b$ in 0...$B$ do
    for $c$ in 0...$C_{in}$ do
        for $f$ in 0...$C_{out}$ do
            for $h$ in 0...$H$ do
                for $w$ in 0...$W$ do
                    Convolve
```

4.3 Fused Kernel

We fuse the depthwise and pointwise operations into one operation, removing the need for allocating space for an intermediate tensor to store the results encouraging better data reuse. However, fusing the depthwise and pointwise kernel does not lend itself to blocking using the LP (mentioned below) and OpenMP synchronization issues. Hence, we do not use it in our final code. [3]

4.4 Loop Unrolling

We explore loop unrolling in our depthwise convolution operation to take advantage of instruction-level parallelism due to pipelining. In addition, we use temporary values to reduce writes to memory. In the inner-most loop that iterates over the filter width, we explicitly calculate k elements per write instead of 1. We create 3 different versions of this unrolling in which the program will jump to depending on the dimensions of the filter. We were able to attain a 1.5x speedup using this optimization.

4.5 Blocking

To optimize cache usage and computation intensity, we utilize blocking in our loops. Blocking our code entails breaking down a large depthwise or pointwise operation into smaller operations of the same type. This increases cache efficiency, and hence performance. This is because when we try to access our data element by element (i.e. without blocking), we have to access slow memory which increases communication time and reduces performance. However, when we block our code, the blocks are pulled into fast memory (L1, L2 cache). This significantly reduces the amount of time it takes for data to reach the CPU to be further processed. Therefore, having a blocked implementation for our code was essential given the nature of the problem and input sizes.

4.5.1 Linear program to obtain tilings

We tried two linear programs [2] [1] to compute tilings that minimize communication. After defining the projective accesses into our tensors, we set up the LP (Dinh et al.) as follows where M is the number of words fit in the L1-Cache and x is log of the tile size :

$$\max_{b} \quad 1^T x$$
$$\text{s.t.} \quad \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} x \leq \begin{bmatrix} 1 \\ 1 \\ 1 \\ \log_M B \\ \log_M C_{in} \\ \log_M W_{out} \\ \log_M H_{out} \\ \log_M W_f \\ \log_M H_f \end{bmatrix}$$

This LP yielded tilings that indicated that we need not block across filters and channels (for small channels) and to block only across batch width and height. This reduced our OpenMP runtime by 2x.

We also tried using the LP in equation 6 in Chen et al to obtain tilings for convolutions. However, the paper uses a small filter trick which slows down performance for our depthwise convolution kernel.

4.5.2 Running a Parameter Sweep to obtain tilings

Fine-grained grid search over all 6 loop tilings would be slow/infeasible. Based on our LP, we know need tile only over Batch, Width and Height. We run a parameter sweep over the three dimensions using a simple grid search to find out which tile sizes worked best across those dimensions in terms of performance. These tilings generalize well to work with other optimizations, leading us to stick to the tilings obtained in the sweep, leading to a 2x speedup over previous tilings when conbined with all other optimizations.

5 Integration with PyTorch

To enable more comprehensive benchmarking, we create a PyTorch extension using our C++ and CUDA code that can be used as a drop-in replacement for existing PyTorch models.

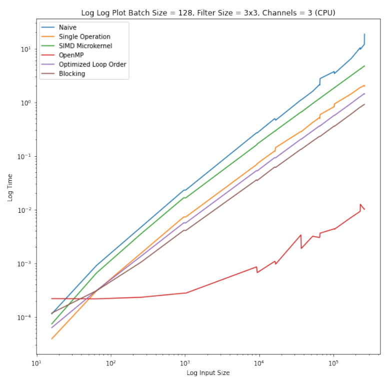
Our implementation uses `setuptools` and `torch.utils.cpp_extension` to compile our back-end code and allow it to be accessed through a Python function call. To translate between PyTorch tensors and the tensors in-memory, we use `torch/extension.h` and the build-in `torch::Tensor` data type.

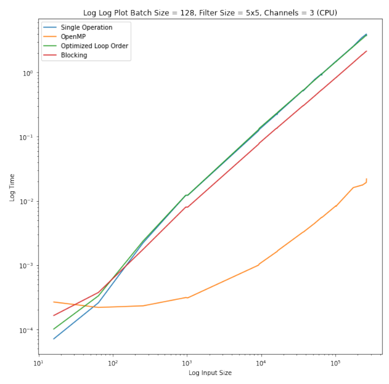
We also develop a PyTorch class that is used similarly to `torch.nn.Conv2D`, except with 2 different filters as it is a depthwise-separable convolution. The constructor calls our `init_conv` function to set up the block sizes, and the `forward` function calls our C++ or CUDA code dynamically depending on the location of the input tensor.[5]

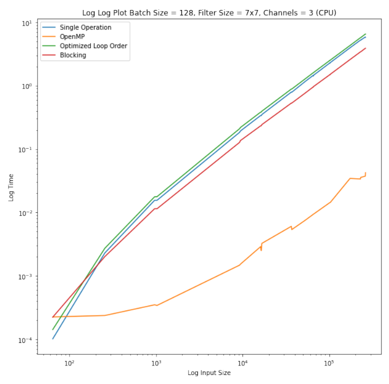
6 Performance Plots

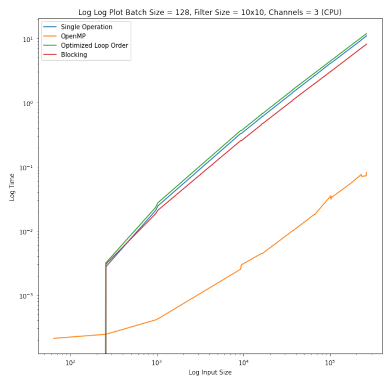
6.1 CPU

6.1.1 Varying Input Sizes









The above plots showcase some general trends. Firstly, there is a clear order where OpenMP performs better than Blocking which performs better than Loop Optimized code which performs better than SIMD which performs better than the Naive code. We can also notice that most of the lines are fairly linear, indicating good strong scaling with larger input sizes. An interesting observation is how OpenMP always curves up. This is likely due to the overhead of creating and running threads for small input sizes which don't need it. However, we can see that eventually, the slope is a lot less steep for OpenMP and it significantly outperforms all other serial code.

6.1.2 Varying Channels

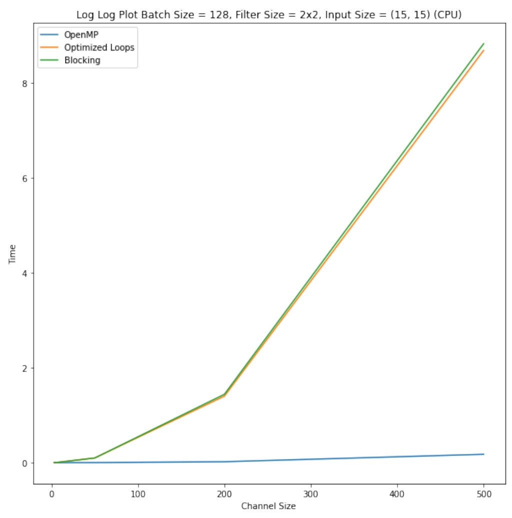


Figure 3: Plot of Channel size v/s Time for varying Channels

The above graph showcases how the timing explodes with larger channel sizes for serial code but how OpenMP handles it fairly well. A big part of this might be due to the storage scheme picked (i.e. (B, C, W, H)). This makes it harder to optimize over channels and makes access a little more difficult. However, in general the scaling is quite linear. Note that we have a drop in the last figure since we skip the small input size since the filter size is bigger than the input tensor.

6.2 CUDA

The plot below shows CUDA runtime on a shared node on Bridges. Note that there are significant fluctuations due to running benchmarking on a shared node. However, it achieves a 2-100x over serial code depending on the input size.

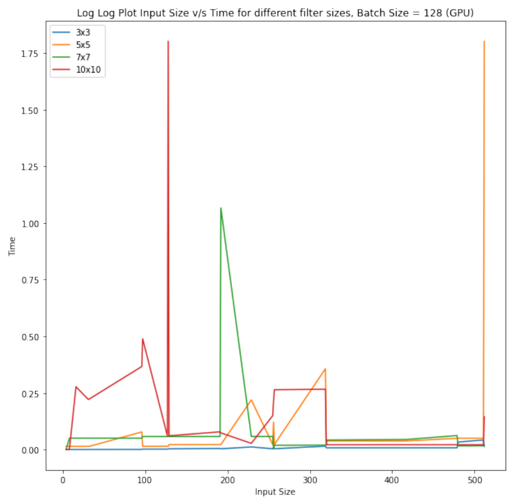


Figure 4: GPU performance on varying input sizes (Input Size v/s Time)

6.3 PyTorch Benchmarks

On a single layer for an input of $128*512*512*3$ our performance (using the Python wrapper) is 0.1377 seconds on CPU. PyTorch in comparison takes 1.264s for the same input. This is a **9.179x** speedup over PyTorch on inference. For GPUs we achieved a 1.1x speedup over PyTorch.

Using our layer in MobileNetV1, we achieve an inference speedup per batch of roughly 2x for a batch of 128 images from ImageNet. As batch size grows, figure 4 demonstrates performance on MobileNetV1.

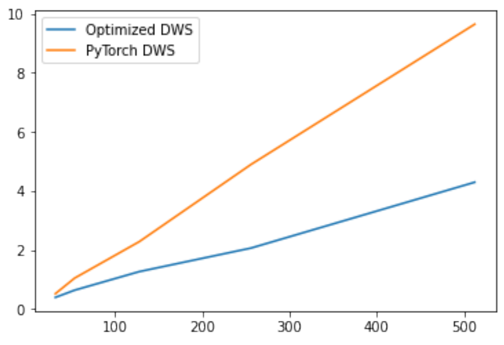


Figure 5: MobileNetV1 inference time for a single batch on CPU

For ConvNeXt, the channel dimension is roughly $O(10^2)$ which is much larger than we anticipated across channels while designing our algorithm's tensor storage. Hence it is a bottleneck and leads to a 2.8x slowdown. To fix this, one can store tensors in B, W, H, C format with channels contiguous.

7 Future Work and Optimizations

There are a few things which we could implement in the future to further speed up and optimize this project. Firstly, we wanted to experiment with trying a new storage scheme to promote parallelism over channels. That would entail switching the current storage scheme from (B, C, W, H) to (B, W, H, C). This would allow us to use OpenMP to parallelize over channels. Next, future work could look at how MPI could be used to obtain data level parallelism and parallelize over batches.

8 Contributions

We worked synchronously in person for a bulk of the project, setting up our core design and code together. Here are some individual contributions in addition to fundamental design:

1. Connor: OpenMP implementation, CUDA implementation, benchmarking infrastructure, PyTorch extension, unrolling optimizations

2. Saurav: Naive implementation and infrastructure, OpenMP implementation, blocking, linear program, microkernel, loop ordering, fused operation, correctness check

3. Tejvir: Naive implementation and infrastructure, Open MP implementation, blocking, linear program, microkernel, loop ordering, fused operation , correctness check

References

[1] Anthony Chen, James Demmel, Grace Dinh, Mason Haberle, and Olga Holtz. Communication bounds for convolutional neural networks, 2022.

[2] Grace Dinh and James Demmel. *Communication-Optimal Tilings for Projective Nested Loops with Arbitrary Bounds*, page 523–525. Association for Computing Machinery, New York, NY, USA, 2020.

[3] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy of high-performance deep learning convolutions on simd architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[5] PyTorch. Custom c++ and cuda extensions¶.