Achieving speedup of convolution operations with applied parallelism

Yongseok Paul Soh
The University of Oregon
Eugene, Oregon, USA
ysoh@uoregon.edu

Joseph McLaughlin
The University of Oregon
Eugene, Oregon, USA
jmclaug2@uoregon.edu

ABSTRACT

Due to the growing complexity and massive amount of parameters required for modern convolution neural networks, computing convolutions in network layers is time-consuming and considered a bottleneck for both the training and testing process. Consequently, there is plentiful ongoing research that aims to speed up these computations. In this work, we investigate and implement three algorithms that are widely used to compute convolutional layers. By implementing the serial implementations as our baselines, we were able to achieve speedups for all algorithms using shared memory multiprocessing and parallel computing APIs such as *OpenMP* and *CUDA*. To validate our implementations, we structured experiments that mimic an actual convolutional layer. We identify interesting characteristics and trends between different algorithms.

This work benefited from access to the University of Oregon high performance computer, Talapas.

1 INTRODUCTION

Recent advances in machine learning are dominated by deep neural networks (DNN). Especially for computer vision tasks, ranging from image classification to image synthesis, convolutional deep neural network architectures are the de-facto standard. The convolutional layers ability to capture localized features within the input data without over-expanding the parameter space is considered as a significant improvement than its previous counterpart, the fully connected layer.

Convolutional layers coupled with deep neural network architecture, allows the model to "understand" complicated representations within the input image as the depth of the network deepens, and constitutes as a key building block for the success of deep convolutional neural network models even to this day. However, the recent advances in convolutional neural network did not come free of price. As complexity of network architectures grew exponentially the amount of computation required has also grown in parallel. Although these augmentations were somewhat manageable by similar improvements of modern SIMD architectures, most importantly, graphical processing units (GPU), the computation of CNNs are still a considerable bottleneck for training and testing CNNs. From our preliminary survey it was surprising to perceive the lack of standardized approaches to convolution computations. We were intrigued to discover that improving the overall speed and efficiency was still an actively researched problem. [4, 10, 12, 13]

For this project, we have implemented convolution computations using three different algorithms that are widely used in practice: direct convolution algorithm, the fast fourier transform (FFT) algorithm and the Winograd algorithm. We use our serial implementations as the baseline for comparison and derive speedups that we observed when implemented using OpenMP and CUDA. [1, 2]

In the following sections, we introduce basic properties of convolution and the characteristics of each algorithm. We provide extensive detail regarding the design and implementation of our serial, OpenMP, CUDA algorithms, as well as our experimental design. Finally, we illustrate our experiments and present our results followed by our analysis.

1.1 Convolution

Convolution is an operation that returns a function j as the result of combining functions two input functions f and g. Intuitively, convolution can be understood as the amount of "overlap" between two functions f, g as g is "shifted" over the domain of f. The result of convolving two functions is then an integral representing this property. Convolution can be understood both *linearly* and *circularly*; linear convolution extends the integral over the complete domain of the function while circular convolution extends the integral over a fixed-period at an arbitrary location.

1.1.1 Discrete Convolution. Several computing applications, including applications of CNNs, image processing, and signal processing, employ a method of convolving discrete series. *Discrete convolution* of two one-dimensional series f and g, involves summing the product of their overlapping indices as g is "shifted" over f, this form follows:

$$(f * g)[n] = \sum_{i=-M}^{M} f[i] * g[n+i]$$

In the context of CNNs—as well as general image processing, discrete convolution is adapted for multidimensional discrete series often taking the form of visual imagery. In two dimensions this can be visualized for two 2D signals d, f by "sweeping" f over d in steps and summing the product of every overlapping pair of elements at every step. **Algorithm 1** illustrates two-dimensional discrete convolution of two series A, B with dimensions X, Y and U, V respectively where a final series C with dimensions, X, Y is the result of convolving the two series.

Algorithm 1 *2D Discrete Convolution:* given three matrices A, B, and C, of sizes $X \times Y$, $U \times V$, and $X \times Y$ respectively.

```
for x = 0 to X do
  for y = 0 to Y do
    for u = 0 to U do
      for v = 0 to V do
        C_{x,y} += d_{x+u-U/2, y+v-V/2} * f_{u,v}
      end for
    end for
  end for
end for
```

Direct convolution remains a fairly computationally expensive task. For two discrete 1D series a, b with sizes x, y respectively, the algorithmic complexity of convolving them is $O(xy)$. However, for two discrete 2D matrices A, B with dimensions X, Y and U, V respectively, the algorithmic complexity of convolving them is $O(XYUV)$.

1.2 Fast Fourier Transform

Circular discrete convolution (i.e. concerning periodic functions) can be accomplished by applying the *convolution theorem*. The convolution theorem follows for discrete periodic functions f, g:

$$\mathcal{F}[f * g] = \mathcal{F}[f] \cdot \mathcal{F}[g]$$

where $\mathcal{F}$ denotes a *complex Fourier transformation*, "$*$" denotes convolution, and "$\cdot$" denotes point-wise multiplication. [14] From this the inverse Fourier transformation, $\mathcal{F}^{-1}$ is applied and yields the form:

$$f * g = \mathcal{F}^{-1}[\mathcal{F}[f] \cdot \mathcal{F}[g]]$$

The Fourier transformation is a method of transposing a function between its domain and its *domain in frequency*. In other words, the transform when applied to a function f produces a function $\mathcal{F}[f]$ such that $\mathcal{F}[f](x)$ is the frequency of the value x in f.

There exist several algorithmic implementations of the Fourier transform, many of which are referred to as *fast Fourier transformations* (FFT.) These methods, when applied with the convolution theorem, typically lead to methods of convolution with algorithmic complexity better than direct convolution ($O(n^2m^2)$.)

1.2.1 *Cooley–Tukey.*

There exist several unique FFT algorithms; the exploration and refinement of improved FFT methods remains an active area of research. However, *the Cooley–Tukey* algorithm is among the most well known methods. The Cooley–Tukey algorithm is a divide-and-conquer technique that computes the discrete fourier transformation (DFT) of a discrete series by first computing the DFT of two interleaved subsections of the series. [5] A given result of DFT X_k from a series X of size N can be founds as follows:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k}$$

where $m = n/2$. [5] This operation bifurcates the original series into interleaved *even-odd indexed* subsections. The DFT is computed recursively for each subsection. The term following the summation is often referred to as a *root of unity*, that is, a term when raised to a positive integer n, is 1. This technique yields an algorithmic complexity of $O(nlogn)$ for a 1D discrete series.

1.2.2 *Applications & Constraints.*

Discrete convolution implemented using the convolution theorem (FFT convolution) has inherent algorithmic properties that reveal application areas distinct from direct convolution. FFT convolution is characteristically circular; circular convolution requires the two series be stable and periodic. In practice, circular convolution is invoked on non-periodic images and from this there exists methods determining "safe regions" within a circular convolution where the result can be presumed accurate. This property permits non-periodic images to be zero-padded, incurring some computational overhead while allowing for accurate results using circular methods.

FFT convolution uniquely requires that the two series being convolved are the same dimensions. This requirement had lead to several different techniques to accomplish the task of convolving discrete series of unequal dimensions; techniques include zero-padding from different sides of the series, evenly padding the series, up-scaling the series, among others. [8]

Further, the Cooley–Tukey algorithm requires that size of the series be a power of two.

One dimensional FFT convolution for two series of length n has an algorithmic complexity of $O(nlogn)$ when implementing Cooley–Tukey. However, the Cooley–Tukey algorithm can be applied to accomplish multidimensional FFT convolution. To accomplish this, we take advantage of the separability property of DFT. That is, to accomplish multidimensional DFT, it is enough to take the DFT of every dimensional row/-column. This is illustrated by **Algorithm 2**. This method for 2D discrete convolution yields an algorithmic complexity of $O(n^2logn)$ to convolve two square $n \times n$ matrices. While this complexity is certainly better than the equivalent $O(n^4)$ complexity of direct convolution, it suits a different application area. For example, when the two input matrices diverge in size, useful work decreases while work remains constant. Further, extra work is required to upscale the input matrices and downscale the output.

Algorithm 2 *2D FFT Convolution:* given three matrices, D, F, and R, all of which have the dimension $N \times N$.

for *row* **of** D **do**
  dft(*row*)
end for
for *column* **of** D **do**
  dft(*column*)
end for
for *row* **of** F **do**
  dft(*row*)
end for
for *column* **of** F **do**
  dft(*column*)
end for
for i **in** *size* **do**
  $R_i = D_i \times F_i$
end for
for i **in** *size* **do**
  R_i = conj(R_i)
end for
for *row* **of** R **do**
  dft(*row*)
end for
for *column* **of** R **do**
  dft(*column*)
end for
for i **in** N **do**
  R_i = conj(R_i)
end for

1.3 Winograd Algorithm

Similar to the FFT algorithm, the Winograd algorithm introduces a set of pre-transforms and post-transforms to reduce the number of multiplications of the actual convolution computation. It is based on the minimal filtering algorithm first proposed by Winograd [15]. The key concept behind the Winograd algorithm is that by applying certain transformations, the identical computation can be performed with a lesser number of computations.

To illustrate this characteristic, we give an example for the convolution of two 1d sequences. Let's say that we are trying to convolve two sequences $[d_0, d_1, d_2, d_3]$ and $[f_0, f_1, f_2]$. The convolution results of these two sequences are $[d_0f_0 + d_1f_1 + d_2f_2, d_1f_0 + d_2f_1 + d_3f_2]$. Note that among the various "modes" of convolutions, for this example we illustrate the "valid" mode where no extra padding is being added to any of the sequences. This convolution computation requires in total, 6 multiplications. The Winograd algorithm, however, reduces the number of multiplications by introducing following transforms:

$$Y = A^T[(Gf) \circ (B^T d)]$$

where Y is the output, f and d are respectively the input sequences and $\circ$ is the element-wise product. A^T, G and B^T are transformation matrices where each have matrix form as following:

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}.$$

$$Gf \text{ is } \begin{bmatrix} f_0 \\ \frac{1}{2}(f_0 + f_1 + f_2) \\ \frac{1}{2}(f_0 - f_1 + f_2) \\ f_2 \end{bmatrix} \text{ and } B^T d \text{ is } \begin{bmatrix} d_0 - d_2 \\ d_1 + d_2 \\ -d_1 + d_2 \\ d_1 - d_3 \end{bmatrix}$$

Since each by-product Gf and $B^T d$ have 4 operands, the actual convolution of the two input sequences requires 4 multiplications instead of 6. Note that for this example, we provided an example for where the input sequences were each of length 4 and 3, and the output length was 2. These

set of transformations are specific to this exact case and for different lengths of input sequences a different set of transformations need to be induced. These transform matrices can induced using the Chinese Remainder Theorem [6].

Theoretically, the larger the input sequences, the more number of multiplications can be reduced. However, the required transformations A^T, G and B^T also increase quadratically in size. Because applying transformations is a form of matrix multiplication, the computation cost for applying these transformation soon outweigh the benefits of the reduction in multiplications for larger input sequences.

2 METHODOLOGIES

We sought to explore the properties of the three methods of convolution discussed for their available performance improvements through two different parallelization techniques. Namely, the three algorithms were parallelized using *OpenMP* and *CUDA*.[1, 2] This section details the methods we derived to optimize these operations and to test their performance.

2.1 Algorithmic Design

The design and implementation of algorithms is critical to their speed in performance, both in serial and in parallel. Each method of convolution that we explored exposed unique properties that we were able to take advantage of in-order to achieve performance speedups.

Every 2D convolution operation we created follows the function signature `void convolve_2d(convolve_2d_t *op)` where
`convolve_2d_t` is defined as follows:

```
/* convolution_types.h */
...
typedef struct {
    float *data,
          *filter;
    size_t data_size_x,
           data_size_y,
           filter_size_x,
           filter_size_y;
    float *result;
} convolve_2d_t;
```

where `data` is a pointer to an array with `data_size_x` × `data_size_y` floating point values, `filter` is a pointer to an array with `filter_size_x` × `filter_size_y` floating point values, and `result` is a pointer to an array with `data_size_x` × `data_size_y` floating point values. Contiguous allocation in this way may improve the performance with regard to memory operations, however the topic is far too dynamic to make broad conclusions. We chose to align our memory this way with the goal of incurring fewer paging lookups, however we consider this simply "best practice."

2.1.1 Direct Convolution. Direct linear convolution as we have implemented follows very closely to **Algorithm 1**. Our initial implementation involved four nested `for` loops, one for each dimension of each matrix. Improving upon this design, we unrolled two of the `for` loops, resulting in only two nested `for` loops. This was a natural first step due to the memory layout we had already selected. In fact, the final design we selected iterates over the entirety of `data`, then `filter` in the inner loop. To accomplish this we calculate the indices of the arrays representing two-dimensional matrices of the current indices in each `for` loop. From here we validate the indices with a conditional before adding it to a rolling sum that appended to the result at the end of the inner-loop.

To parallelize this implementation in OpenMP we settled on applying a `#pragma parallel for` to the outer loop. [2] For two matrices of size $n \times n$, $m \times m$ this implementation means that each thread works one cell of the result matrix in parallel. Attempts were made to parallelize the OpenMP version further with nested parallelism, but these designs were not effective for the task. [2] Under conditions when `filter` was small it did not benefit the design to create more threads.

We parallelized our CUDA implementation with the same approach; initially we sought to divide the operation's work such that one result cell of the `result` matrix would be computed by one CUDA thread. We explored computing the `result` matrix in part, using atomic operations to sum the value, but found that this complicated the design and did not lead to immediate performance improvements. We chose to statically define a CUDA block size and to divide it by the size of `result` to determine the number of blocks needed.

2.1.2 FFT Convolution. We implemented FFT convolution following the convolution theorem. We began by implementing radix-2 Cooley-Tukey DFT. Our initial implementation was computed recursively and allocated a temporary array of size $N \times logN$ for an array of length N. We modified this design later by computing it iteratively instead and only allocating an array of size $2N$, swapping pointers with each loop iteration. While this reduced the spacial complexity, it made this portion of the algorithm more difficult to parallelize without modifying the design. A notable optimization we accomplished was a "blanket" unrolling of the interleaved DFT operation. Instead of having a two `for` loops, one to iterate through the number of DFTs in the current layer and another to compute inside on the individual DFTs, we were

able to to unroll this loop. In total, we were able accomplish DFT using two `for` loops, one for the radix-2 decimation ($O(logn)$) and another for every element in the array ($O(n)$.)

Here, it is necessary to clarify that FFT covolution internally requires complex arithmetic. We used the C99 standard library `complex` type specifier and its associated functions defined by `complex.h`. However, because the inputs and outputs are `float` arrays without a `complex` type specifier we incur some penalty when converting to and from `float complex`. This is included in our performance measurements of this implementation. From here, we follow the form of **Algorithm 2** when implementing FFT convolution.

To parallelize our implementation in OpenMP we first attempted to parallelize the DFT operation. [2] We explored parallelizing the inner-loop contributing to $O(n)$ algorithmic complexity, but we found more success applying parallelism to the loop making the individual DFT calls itself. We attribute this to fewer context-switches and better saturation of work.

2.1.3 Winograd Algorithm. To apply the Winograd algorithm so that it can perform a generic convolution operation for 2D images, we need to consider the following: First, we need to expand the algorithm to be applicable for 2D sequences. This can be done by nesting the 1D algorithm. The resulting computation is as follows:

$$Y = A^T[(GfG^T) \circ (B^T dB)]A.$$

As previously mentioned in **Section 1.3**, it doesn't make practical sense to apply these transforms for cases where f and d are both large. However, when at least one input sequence has a small size ($n \leq 3$) the Winograd algorithm becomes applicable. In the single dimensional convolution operation point of view, the length of the filter determines the length of the receptive field on the input. That is to say, even when the input is a arbitrarily larger size, by slicing it into smaller pieces that match the receptive field of the filter, we are able to apply this algorithm.

For implementation, it is therefore necessary to slice the input sequence into smaller subblocks. The size of the subblocks is dependant upon the specific variation of the Winograd algorithm. For our project, we use filter size 3×3 and subblock size 4×4. Similar to the example shown in **Section 1.3**, this operation results in a output of size 2×2. The same variation was used in [10], where it first introduced the Winograd algorithm for convolutional computations in CNNs.

Slicing the input into 4×4 subblocks introduced a couple of implemental issues. One of which was that for every 4×4 subblock, the output produced a 2×2 block. Implementing this straightforwardly would result in the output being half the size of the input. To avoid this issue, we sliced the subblock so that it would overlap the neighboring subblock by a index of 1 on all sides. Because the output block was of size 2×2, the corresponding input subblock would be a 4×4 block that surrounded the 2×2 block with 1 elements of padding on the top, bottom, left and right. This allowed the output to have an identical size as the input. The other issue was caused by this overlapping scheme. On the edges of the input, the subblocks to be parsed were out of index. In this case, we filled the out of index regions as zeros so that it can still be of size 4×4. Based on this subblocking scheme we were able to deterministically split the input with size $H \times W$ into $\lceil H/2 \rceil \times \lceil W/2 \rceil$ subblocks.

The following step is to apply the Winograd transformation to the data subblocks and the filter, GfG^T, $B^T dB$ respectively. This maps each operands to 4×4 matrices which is then used for element-wise multiplication $GfG^T \circ B^T dB$. It is interesting to note that even up to this point, the actual mixing of data and filter is not occurring. This allows all operations up to this point to be executed without any race conditions.

Once the element-wise multiplication $GfG^T \circ B^T dB$ is executed, the final step is transforming the 4×4 by-product to the final 2×2 output. This is done by multiplying the 4×4 matrix with a 2×4 A^T and 4×2 A matrix sequentially.

Because Winograd algorithm uses a "divide and conquer" approach, parallelizing our implementation in OpenMP was relatively straightforward. However, some design details were taken into consideration. Because the prerequisite transforms for the subblocks and filter were independent from the element-wise multiplications and post-transforms, our initial attempt was to spawn parallel regions for each step of the process. Empirically, however, we discovered that wrapping the entire sequence for a single subblock as a subroutine and spawning threads for each subroutine was a superior approach. For our CUDA implementation, we initially copied all input and filter data to the GPU and rather than implementing the entire sequence into a single kernel, we implemented each functional block as its own kernel.

2.2 Experimental Design

To effectively profile our algorithmic designs, we developed modelled two of our our experimental groups against key aspects of CNNs. Over the past decade, various types of convolutional layers were introduced and used in CNN models. For the relatively early *AlexNet* type networks[9], various

filter sizes were used ranging from 11×11 to 3×3. More recently, *ResNet* was introduced where only a combination of 3×3 and 1×1 size filters were used alongside residual connections between layers [7].

To measure the performance applicable to CNNs, we developed two experiments modelling two distinct features of CNNs: the *Input layer*-type and the *Feature layer*-type. We further model performance by examining various *filter sizes* in one final group.

2.2.1 Input Layer. All CNN models have an input layer. They serve the purpose of feeding the raw sequences, such as images, into the model for further feature representation. In earlier CNN architectures, larger filter sizes were used in these layers. The idea behind it was to provide a feature representation to the next layer that had a bigger receptive field. In other words, it was considered to be intuitively better when a element in the next layer was a mixture of more signals from the input layer.

For our experiments, we simulate an input layer type computation to test our implementations performance. We use both real images and synthetic images with the size of 256×256, 512×512 and 1024×1024 pixels. All have RGB channels. We also vary the filter sizes from 3×3, 5×5 and 7×7.

2.2.2 Feature Layer. Further into a typical CNN model architecture comes a feature layer. Traditionally, it can be also called as the hidden layer. The structure of the feature layer is drastically different from the input layer. While input layers have larger height and width dimensions with small number of channels, typical feature layers have smaller height and width sizes and have substantially larger number of channels. For the recently prevalent *ResNet* models, the height and width range between 56×56 to 7×7 while the channels range from 64 to 1024.

To test how each of our implementations performed on these type of computations, we mimicked the feature layer by fixing the height and width to 16×16 for the input and 3×3 for the filter and varied the number of channels for the input and filter from 256, 512, 1024.

2.2.3 Comprehensive Trials. The two types of convolutions computations required in a CNN are similar in a way that the sizes of two sequences are significantly different. For convolution computations required in input layers, the size of the input can be hundreds of thousands times larger than the size of the filter. Even for feature layers, the height and width of the feature "tensors" can be up to 30 times larger than the size of the filter. We thought it would be worth investigating the performance of each implementations when this was not the case.

For this purpose, we designed a comprehensive experiment that tested each implementations performance for various input to filter size ratios. For the experiments, the input size was fixed to 512×512 while the filter size ranged from 2×2 to 512×512.

3 RESULTS

We evaluated our designs on Talapas. [3] Our code ran on one node using the 28 available cores of the *E5-2690v4* processor one *NVIDIA Tesla K80* GPU accelerator. We measured runtime using clock-cycles determined by the `TSC` instruction. The results of all of our convolution operations measured runtime beginning with allocated and initialized memory for the inputs and outputs and ending when the result of the operation is copied back. We validate our results by comparing the results of each operation against a baseline function. For most operations this baseline is the direct serial implementation, however because FFT convolution is circular, results for FFT are compared against the FFT serial implementation.

3.1 Input Layer

For input layer experiments, we used real and synthetic images with the sizes of 256×256, 512×512 and 1024×1024. All have three channels that represent the RGB colors. In actual CNN models, the weight of the filters are learned through backpropagation [11]. For our experiments, however, we hand-coded in some simple filters to validate our implementations. We used two types of filters: a 2D Gaussian blur filter and 2D second derivative filter. Figure 1 shows the convolution results for these filters and images.

For quantitative comparisons, **Figure 2** exhibits the runtime for all implementations. Table 2 shows the actual runtimes for all implementations. The prefix *D*, *W* and *FFT* stands for the type of algorithm; direct, Winograd and FFT. The postfix *S*, *OMP* and *CUDA* represents the implementation method, serial, OpenMP and CUDA. [1, 2]

For serial implementations, results show that throughout all input sizes, the Winograd algorithm showed the best performance. For input size 1024×1024, the Winograd algorithm was up to 60% faster than the direct implementation. The performance of the FFT implementations, however, was significantly slower than the other two implementations. This can be accounted by the characteristics of the FFT algorithm. Since the algorithm requires both the input and filter to be in identical sizes, when the filter is relatively smaller than the input, it causes an unnecessary computational overhead that results in longer runtime. More details on this characteristic will be explained in **Section 3.3**.

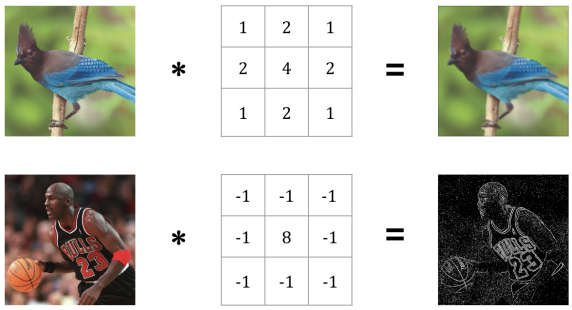

Figure 1: Filters and input-output images from input layer experiment

	Implementations			
Input size	D-S	D-OMP	D-CUDA	W-S
256×256	0.0185	0.0188	1.50E-05	0.0117
512×512	0.0741	0.0774	1.63E-05	0.0475
1024×1024	0.2960	0.3129	3.03E-05	0.1864
Input size	W-OMP	W-CUDA	FFT-S	FFT-OMP
256×256	0.0025	0.0008	0.8914	0.0435
512×512	0.0091	0.0011	4.0128	0.1745
1024×1024	0.0387	0.00251	7.8287	0.7846

Table 1: Runtime(seconds) comparison for real images with 3×3 filter convolutions

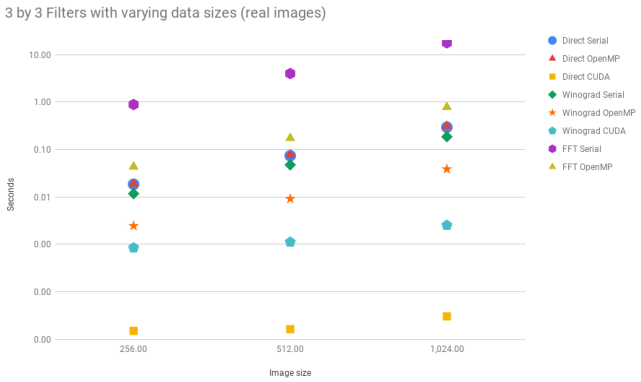

Figure 2: Runtime for real input images with size 256×256, 512×512 and 1024×1024 using 3×3 filters

For OpenMP implementations, the performance gain the Winograd algorithm was able to achieve was even more impressive. For input size 512×512, the Winograd implementation was 8.46 times faster than the direct algorithm. For FFT implementations, despite still being slower than

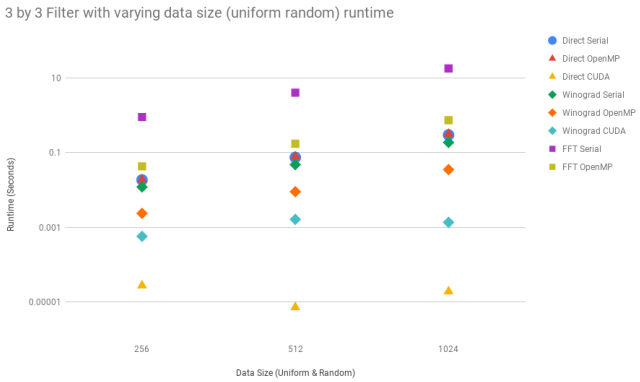

Figure 3: Runtime for synthetic input images with size 256×256, 512×512 and 1024×1024 using 3×3 filters. (Lower is better)

direct and Winograd, we can see compared to the serial implementation there was a 20 times speed up.

The results for CUDA implementations were interesting considering that the direct implementation outperformed the Winograd implementation. While the Winograd still gained up to 9 times speedup compared to OpenMP implementations, the runtime of direct implementations were up to 10,000 times faster. Counter-intuitive at first, we believe this can be attributed to the difference in number of threads available in CUDA. Because the convolution computations can be split into smaller parallelizable "work" than the Winograd algorithm, it was able to achieve speedups in orders of magnitudes.

Figure 3, Figure 4, Figure 5 show comparative experiment results for filter sizes 3×3, 5×5 and 7×7 with synthetic input images. Although exact performance gains may slightly vary, the characteristics of the results were consistent with our previous analysis. Note that the Winograd implementation results are only included in experiments using 3×3 filter size.

3.2 Feature Layer

One of the key differences in convolution computations between feature layers and input layers is that it has smaller height and width dimensions but larger number of channels. All experiments used input size 16×16 and filters with size 3×3 while the number of channels varied from 128 to 1024.

In **Figure 6** we show the runtimes for serial implementations for all three algorithms. We can see that for all data arrangements, the Winograd algorithm outperforms both

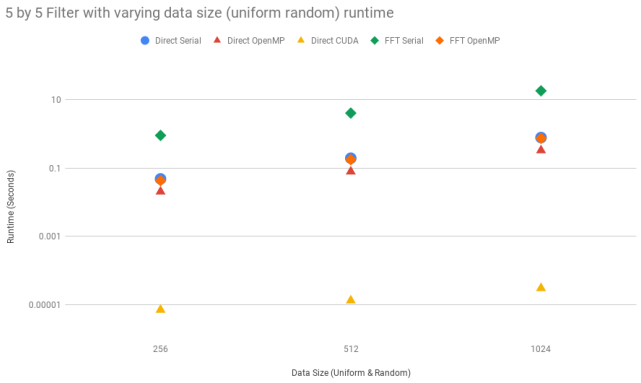

Figure 4: Runtime for synthetic input images with size 256×256, 512×512 and 1024×1024 using 5×5 filters

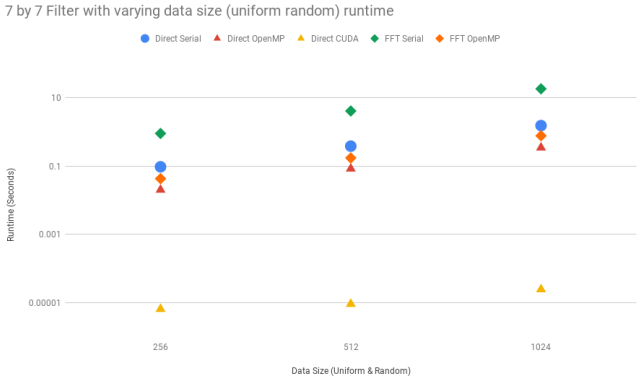

Figure 5: Runtime for synthetic input images with size 256×256, 512×512 and 1024×1024 using 7×7 filters

the direct and FFT algorithm. For the case of $16 \times 16 \times 512$, Winograd implementation shows 60% speedup compared to direct implementation. Also, unsurprisingly, we can observe that as the data arrangement gets larger, the runtime increases quadratically for all implementations.

In **Figure 7** we plot the runtimes for OpenMP implementations for all three algorithms. The relative runtimes are similar to their serial counterparts, where Winograd outperforms direct implementation by a factor of 2. Interestingly, however, the absolute runtimes are orders of magnitudes slower than the serial versions. Our interpretation of these results is that the experimental design of the feature layer causes too much context-switching to allow for effective parallelism; while the input layer passed a relatively small filter (e.g. 5×5) over a relatively large images (e.g. 512×512,) the feature layer passed the same small filter over hundreds

	Implementations			
Input size	D-S	D-OMP	D-CUDA	W-S
128	0.00309	0.04661	0.00018	0.00195
256	0.00613	0.09263	0.00034	0.00389
512	0.01241	0.18552	0.00067	0.00775
1024	0.02449	0.37178	0.00133	0.01550
Input size	W-OMP	W-CUDA	FFT-S	FFT-OMP
128	0.02241	0.00063	0.00305	0.04630
256	0.04503	0.00115	0.00619	0.09275
512	0.08971	0.00234	0.01218	0.18573
1024	0.18055	0.00513	0.02440	0.37150

Table 2: Runtime(seconds) comparison for feature layer convolution computations

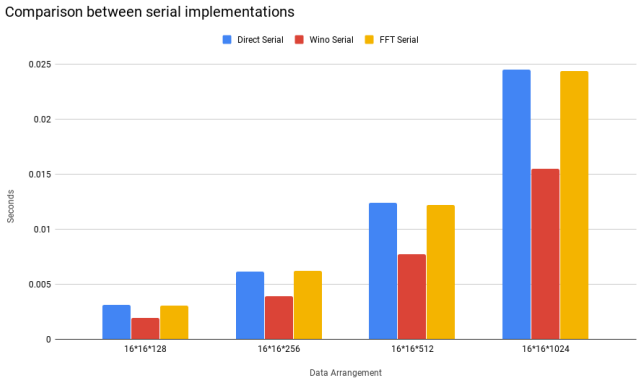

Figure 6: Runtime comparison for serial implementations for feature layers with size $16 \times 16 \times 128$, $16 \times 16 \times 256$, $16 \times 16 \times 512$ and $16 \times 16 \times 1024$ using 3×3 filters

of small images. For CUDA however, this is not an issue due to the abundance in threads and our experiment results back our assumption by showing the direct and Winograd CUDA implementations attaining up to $\times 17$ and $\times 3$ speedups respectively. In retrospect, by further optimizing the implementations for OpenMP alongside the data access patterns, we believe it is possible to have OpenMP implementations with better performance. Future work could consider designing convolution operations specifically for data arrangements with many small images; a "2.5-D" convolution operation might allow for multiple images to be processed without context-switching. The speedup comparison for direct and Winograd implementation individually are shown in **Figure 8** and **Figure 9**.

3.3 Comprehensive Trials

We performed a block of comprehensive trials for 512×512 matrices where for each trial the filter size was increased

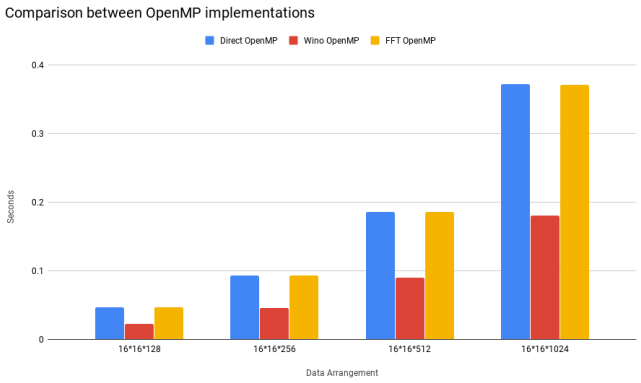

Figure 7: Runtime comparison for OpenMP implementations for feature layers with size $16 \times 16 \times 128$, $16 \times 16 \times 256$, $16 \times 16 \times 512$ **and** $16 \times 16 \times 1024$ **using** 3×3 **filters**

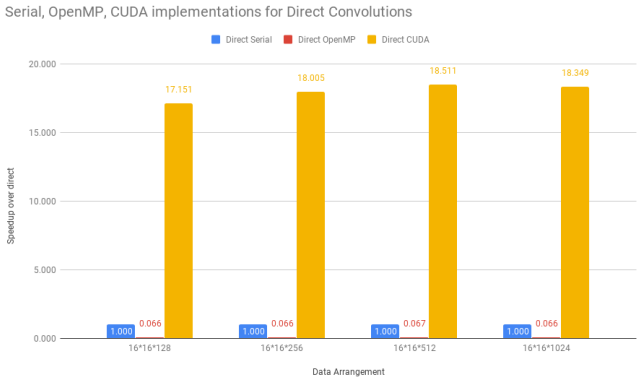

Figure 8: Direct convolution speedup comparison for feature layers with size $16 \times 16 \times 128$, $16 \times 16 \times 256$, $16 \times 16 \times 512$ **and** $16 \times 16 \times 1024$ **using** 3×3 **filters**

by a power of two, starting at 2×2. These results illustrate the complexity of direct convolution; immediately, the direct serial implementation's runtime is multiplied by four when the area of the filter is multiplied by four. Both direct parallel implementations also succumb to this trend, however this delayed to degree in both implementations. This characteristic is illustrated by **Figure 10** wherein the runtime for the OpenMP and CUDA direct serial implementations are depressed until the filter size grows beyond 16×16. For filters larger than 16×16, the parallelized direct serial implementations follows the complexity of the serial implementation. Note however, for **Figures 10, 11**, the vertical axis scales logarithmically, implying a significant difference between the runtimes/speedups of the different implementations.

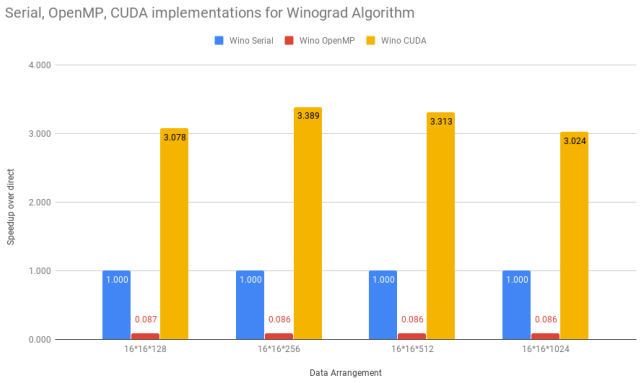

Figure 9: Winograd algorithm speedup comparison for feature layers with size $16 \times 16 \times 128$, $16 \times 16 \times 256$, $16 \times 16 \times 512$ **and** $16 \times 16 \times 1024$ **using** 3×3 **filters**

From **Table 3**, we observed a $\times 20$ speedup using OpenMP and a $\times 72$ speedup using CUDA with a 32×32 filter. However, this speedup becomes less meaningful as the complexity of the operation grows too strongly with the filter size. Under conditions when the filter size is larger, we observed that the FFT implementations were faster than the direct implementations; filter size is not a direct factor in FFT convolution's complexity, however data size is a direct factor. This observation suggests a split problem-space between direct and FFT convolution. For large filters it appears that FFT convolution is better, but for small filters it would appear that the direct method is still faster. We illustrate this bifurcation in **Figure 11** where direct CUDA convolution has the largest speedup on the left of graph (denoting small filter sizes) and the FFT OpenMP convolution implementation has the largest speedup on the right of the graph (denoting large filter sizes.)

Future work could consider how to further optimize FFT in both serial and in parallel. Calculating the root of unity at each step in the Cool-Tukey algorithm is a fairly compute heavy task that might benefit from memoization or pre-computation in certain cases. Otherwise, a modified algorithm that combines row/column traversing `for` loop with the Cooley-Tukey algorithm might benefit parallel implementations by reducing context-switching.

4 CONCLUSION

The increasing complexity of constructing and training convolutional neural networks (CNNs) has stirred interest in improving convolution operations as they are considered a bottleneck to network design. Consequently, there exist numerous ongoing research topics aiming to improve the

	Implementations				
Filter	D-S	D-OMP	D-CUDA	FFT-S	FFT-OMP
2×2	0.0362	0.0811	0.0079	4.0020	0.1813
4×4	0.1281	0.0977	0.0085	4.0065	0.1749
8×8	0.4890	0.0893	0.0127	4.0057	0.1750
16×16	1.9295	0.1249	0.0317	4.0204	0.1748
32×32	7.7966	0.3915	0.1072	4.0056	0.1751
64×64	29.905	1.4204	0.3702	4.0051	0.1881
128×128	115.75	5.4992	1.2086	4.0016	0.1750
256×256	453.95	21.712	4.0967	4.0099	0.2752
512×512	1745.5	83.389	14.634	4.0004	0.1809

Table 3: Runtime (seconds) comparison for synthetic 512×512 images convolved with various filter sizes

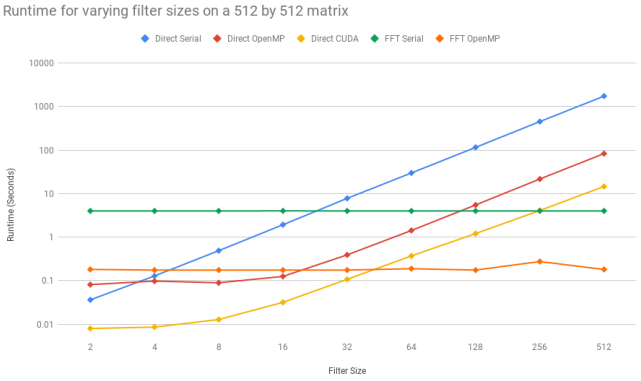

Figure 10: Comprehensive experiment results. Runtime comparison for various filter sizes (lower is better.)

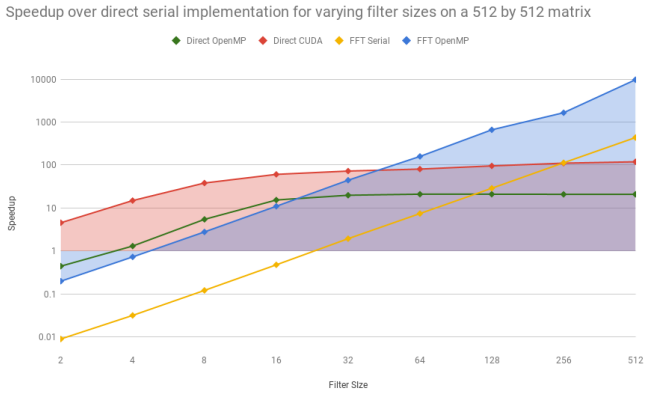

Figure 11: Comprehensive experiment results. Speedup comparison for various filter sizes (higher is better.)

performance of convolutions. We implemented three methods of performing discrete convolution that are widely used in CNNs: direct convolution, fast Fourier transformation, and the Winograd algorithm. We evaluate each implementation for characteristics that enable parallelism. After evaluation, we apply parallelism to the three implementation using both *OpenMP* and *CUDA* parallelism APIs. We share experiments that we constructed to measure useful speedup due to parallelism within the context of a CNN, as well as the direct performance of convolution operations. We demonstrate significant speedups by applying parallelism, for all three methods.

REFERENCES

[1] Cuda documentation page. URL https://docs.nvidia.com/cuda/. Visited on 2019-12-03.

[2] Openmp homepage. URL https://www.openmp.org/. Visited on 2019-12-03.

[3] Talapas homepage. URL https://hpcf.uoregon.edu/content/talapas. Visited on 2019-12-01.

[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. URL http://dblp.uni-trier.de/db/journals/corr/corr1410.html#ChetlurWVCTCS14.

[5] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19 (90):297–301, 1965.

[6] C. Ding, D. Pei, and A. Salomaa. *Chinese Remainder Theorem: Applications in Computing, Coding, Cryptography*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996. ISBN 981-02-2827-9.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[8] Christopher Jekeli. Error analysis of padding schemes for dft's of convolutions and derivatives. Technical report, Ohio St. Univ. Columbus Dept. of Civil and Environmental Engineering, 1998.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[10] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *CVPR*, 2016.

[11] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio. Object recognition with gradient-based learning. In D. Forsyth, editor, *Feature Grouping*. Springer, 1999.

[12] Lingchuan Meng and John Brothers. Efficient winograd convolution via integer arithmetic. 01 2019.

[13] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann Lecun. Fast convolutional nets with fbfft: A gpu performance evaluation. 12 2014.

[14] Eric W. Weisstein. Convolution theorem. URL http://mathworld.wolfram.com/ConvolutionTheorem.html. Visited on 2019-12-01.

[15] Shmuel Winograd. Arithmetic complexity of computations. Siam, 1980.