Fast CUDA Kernels for ResNet Inference

Qiantong Xu
qx57@cornell.edu

Sailun Xu
sx38@cornell.edu

Abstract

In this work, we implement fast cuda kernels for the most time-consuming parts of ResNet – convolution layers of residual blocks with 128 channels and 256 channels. Our kernels[1] are designed for single example inference of ResNet on NVIDIA TITAN X GPUs. With more balanced job allocation among threads, more efficient algorithms and parameters tuned for specific device, we can get up to 50% performance boost than cuDNN.

1 Introduction

We know that *Deep Neural Network* (DNN) dominates the ML models we used in solving problems in different area, because of its incomparable representability. However, with abundant parameters in each DNN model, the training process and, more importantly, the corresponding inference phase become time-consuming. In particular, the bottle-neck is the matrix multiplication (MM) in fully-connected layers and convolution layers. Scientists and engineers have spent decades to improve DNN with better computational efficiency.

The popularity of DNN starts after the advent of GPUs, because GPUs can speed up MM computation tens or hundreds times compared with CPU. Meanwhile, NVIDIA developed a library cuDNN [NVI], specialized for DNN training and inference, which has become the standard of the industry thanks to its robust high performance. The implementation of MM in cuDNN for MM is efficient for any input size on any type of devices (GPUs). Besides, cuDNN is widely used in almost all the major DNN frameworks including caffe, pytorch, tensorflow and mxnet.

The main goal of our project is to develop CUDA kernels for specific convolution computations with performace better than cuDNN. Specifically, we want to develop six types of CUDA kernels for convolution layers in the most popular DNN architecture *ResNet* [He+16]:

1. 1 input feature map ($14 \times 14 \times 512$) with 128 conv-kernels ($1 \times 1 \times 512$)
2. 1 input feature map ($14 \times 14 \times 128$) with 128 conv-kernels ($3 \times 3 \times 128$)
3. 1 input feature map ($14 \times 14 \times 128$) with 512 conv-kernels ($1 \times 1 \times 128$)
4. 1 input feature map ($14 \times 14 \times 1024$) with 256 conv-kernels ($1 \times 1 \times 256$)
5. 1 input feature map ($14 \times 14 \times 256$) with 256 conv-kernels ($3 \times 3 \times 256$)
6. 1 input feature map ($14 \times 14 \times 256$) with 1024 conv-kernels ($1 \times 1 \times 256$)

The reason why we choose to implement these kernels is that they dominate the computation time in ResNet. If we can speed them up, we can gain a lot efficiency when deploying ResNet models in practice. Note that we only focus on optimization for the single example inference, i.e. the input batch size for ResNet is one and there is no training related operations like back propagation. In addition, following the convolution computing, there are Batch-Normalization [IS15], scaling and activation layers. Those layers are always implemented independently with each other by calling different cuDNN functions, but in inference, it is quite reasonable for us to implement them together into a single kernel without accessing memory more times.

In our implementation, we use *Winograd* [LG16] algorithms for 3×3 kernels, because it can greatly reduce the number of floating-point operations in convolution computation. Besides, all the convolution computations are transformed into efficient matrix matrix multiplication (MM). In the following sections, we will discuss more detail about how to implement *Winograd* algorithm and MM efficiently on GPU.

[1] `https://github.com/xuqiantong/CUDA-Winograd`

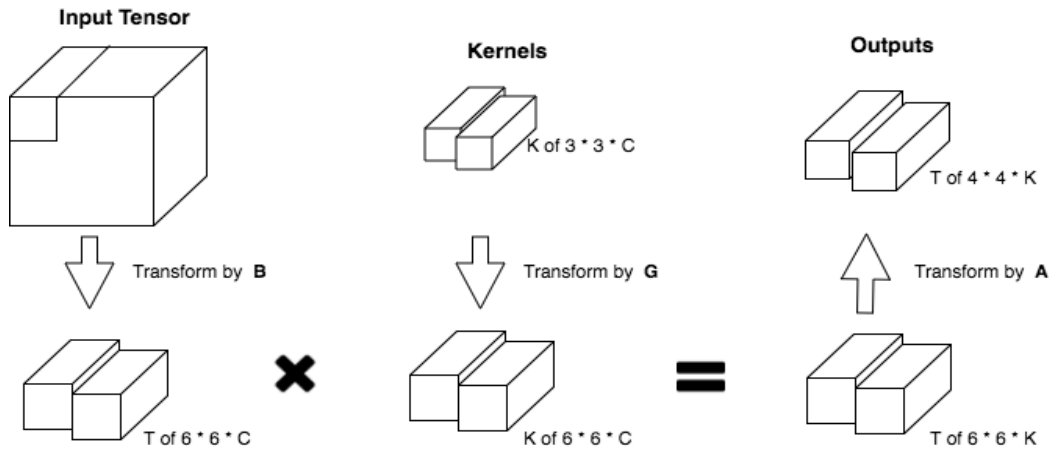

Figure 1: Winograd Algorithm (K is the number of filters/output channels, T is the number of 6×6 tiles of the input)

2 The *Winograd* Algorithm

For the $3\times3\times128$ and $3\times3\times256$ kernels, we use the Winograd Algorithm to reduce the number of multiplication required so as to accelerate the convolution computing. The version of the algorithms that is applied to our kernels is described below.

2.1 One Dimensional Case

Suppose now we have one dimensional input signal, i.e. width and the number of channels are both 1. We use $F(m, n)$ as the notation for computing m outputs using kernel of size n. Then we have the following algorithm for $F(4, 3)$

$$F(4,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \\ d_2 & d_3 & d_4 \\ d_3 & d_4 & d_5 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = A^T[(Gg) \odot (B^T d)] \tag{1}$$

Where

$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & 2 & 1 & 2 & 1 & 0 \\ 0 & 2 & 1 & 2 & 1 & 0 \\ 0 & 4 & 0 & 5 & 0 & 1 \end{bmatrix}, G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix}, A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}$$

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T, \quad d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 & d_4 & d_5 \end{bmatrix}^T,$$

g is the convolution kernel and d is the input. In doing this, we could reduce the original 12 multiplications to only 6 (input transformation is not taking into account).

2.2 Two Dimensional Case

If the input signal is two dimensional, whose both height and width are larger than one, we could nested the one dimensional algorithm to get two dimensional algorithm

$$F(4 \times 4, 3 \times 3) = A^T[(GgG^T) \odot (B^T dB)]A \tag{2}$$

In this two dimensional case, we could reduce the original $12^2 = 144$ multiplications to only $6^2 = 36$ ones.

2.3 Three Dimensional Case

Now if the input is a three dimensional tensor, which is exactly what we are dealing with in ResNet inference, we can further extend the two dimensional algorithms following the 3-D convolution by computing convolution

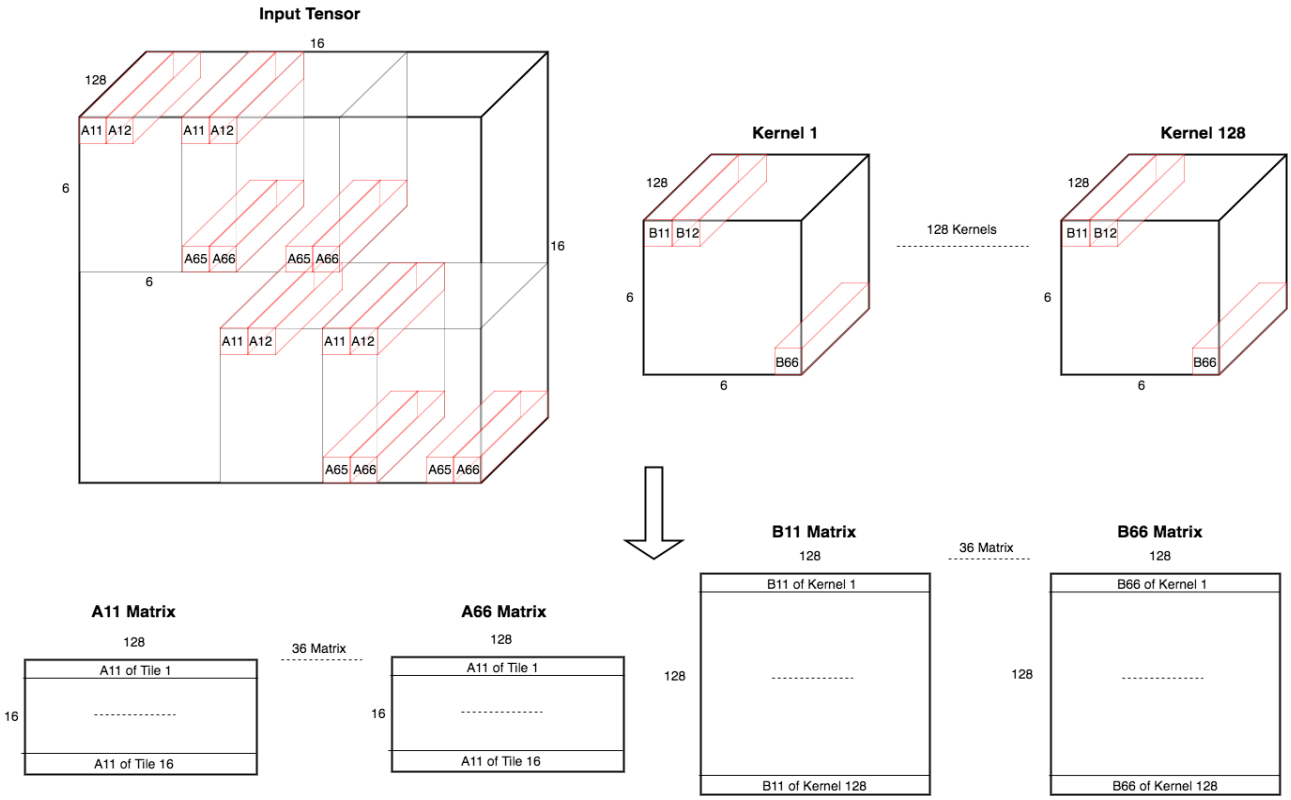

Figure 2: Apply matrix multiplication on Winograd Algorithm. Using the second task as an example. Due to the 1-padding used in ResNet, we change the input size from 14 to 16.

independently for all the channels and summing up the results.

$$F(4 \times 4 \times C, 3 \times 3 \times C) = A^T \Big[\sum_{i=1}^{C} (G g_i G^T) \odot (B^T d_i B) \Big] A \tag{3}$$

The whole process is shown in Figure 1. In essence, we first transform the $3 \times 3 \times C$ kernel into $6 \times 6 \times C$ kernel using $g' = GgG^T$ and similarly transform the input into different $6 \times 6 \times C$ tiles using B. Then, element-wise multiply the transformed input and kernels together and sum them up for all the channels (along the third direction) to get different 6×6 matrices. Finally, we can get the output 4×4 tensor by transform the 6×6 matrices by A and concatenate them together along the third direction. If we have a bank of K filters, then the output would have channel number of K.

2.4 Apply matrix multiplication on *Winograd* Algorithm

As the total number of multiplication required is a constant for each convolution layer, therefore if we could do more computation given the fixed amount of data loaded into the shared memory, we can make the whole computation more efficient because memory access costs much more than doing computation. Thus, instead of naively take out $6 \times 6 \times C$ of data at a time and compute the convolution with the transformed $6 \times 6 \times C$ kernel (n^2 data with n^2 computation), we instead do the following trick. As shown in Figure 2, we notice that we are computing the convolution of a grid of input against the same part of kernels. For example, the left-top corner of each input kernel ($B11$) will only interact with the left-top corner ($A11$) of each 6×6 tile of the input tensor and similar pattern can apply to elements in other positions (Aij and B_{ij}). This way, we can form 36 distinct matrix multiplication for this convolution computing (n^2 data with n^3 computation). If we still use the example in Figure 2, each of the 36 MM can be formed as follow:

$$C_{ij} = A_{ij} B_{ij}^T = \begin{bmatrix} A_{ij}^{(1)} \\ \dots \\ A_{ij}^{(16)} \end{bmatrix} \begin{bmatrix} B_{ij}^{(1)} \\ \dots \\ B_{ij}^{(128)} \end{bmatrix}^T \tag{4}$$

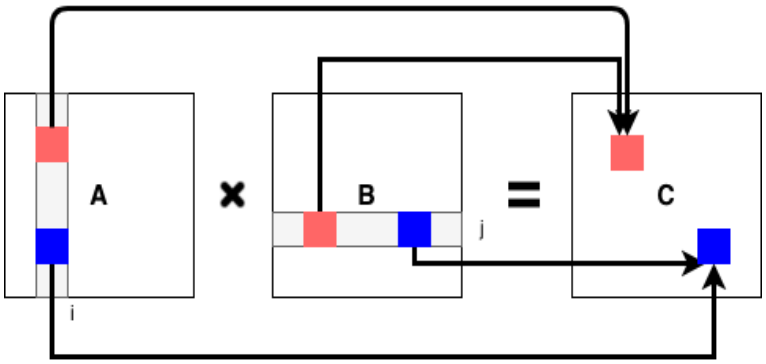

Figure 3: Using outer product to implement matrix multiplication

Where A_{ij} is the matrix of the elements on row i and column j of all the input tiles and B_{6*i+j} are the elements A_{ij} will interact with from all the kernels. We therefore transform the traditional convolution computing into MM which can better utilize the power of GPU.

3 Matrix Multiplication on GPU

3.1 Blocking

As we have done extensively before, we again first partition the matrix into some components we could calculate independently:

$$\begin{bmatrix} \boldsymbol{A_1} \\ \boldsymbol{A_2} \\ \boldsymbol{A_3} \\ \dots \\ \boldsymbol{A_N} \end{bmatrix} \begin{bmatrix} \boldsymbol{B_1} & \boldsymbol{B_2} & \boldsymbol{B_3} & \dots & \boldsymbol{B_M} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A_1 B_1} & \boldsymbol{A_1 B_2} & \dots & \boldsymbol{A_1 B_M} \\ \dots & \dots & \dots & \dots \\ \boldsymbol{A_N B_1} & \boldsymbol{A_N B_2} & \dots & \boldsymbol{A_N B_M} \end{bmatrix} \tag{5}$$

Where we partition the first matrix by rows and second matrix by columns. The reason of conducting blocking MM is that we want to minimize the global memory access and maximize the utilization of the shared memory. In particular, the A_i and its corresponding out put sub-matrix will be held in the shared memory through the whole computing process while we will iteratively fetch part of B_j into shared memory and do the computing of it with A_i. Furthermore, we have tuned N and M according to the GPU we are using to get optimal performance. All the details are listed in the next section.

3.2 Outer Product

In calculating the matrix multiplication that is essential for both the $1 \times 1 \times C$ and $3 \times 3 \times C$ case, we use outer product instead of inner product. The reason is that inner product is not optimal for parallel programming. For example, suppose we have to calculate the following matrix multiplication: $A \times B = C$, then in inner product we will compute:

$$C[i][j] = \sum_k A[i][k]B[k][j] \tag{6}$$

We can compute all the multiplications in parallel but not the summation. Even though we can use reduce sum for efficiency, there will still be a bunch of idle thread. This will cause great efficiency loss due to workload imbalance. However, replacing the inner product with outer product is a effective way to alleviate this issue:

$$C^{(k)} = A[:][k]B[k][:], C = \sum_k C^{(k)} \tag{7}$$

In calculating A[:][k]B[k][:], every thread would only be responsible for multiplication between an entry of A and an entry of B, and every thread could independently write to result C by adding the current entry at C with the new value it has just calculated. The matrix C will only be written to memory once all the computing is done, namely we only write out once for the all final results.

4 Implementation Details

Aside from developing model-specific CUDA kernels, we are also trying to make our kernels device-specific. This means we need to tune the parameters for the given algorithm in each case based on the device it runs on in order to maximize the utilization and occupancy. Since we are developing our kernels on NVIDIA TITAN X, there are four main limits need our attention – 1) **number of Streaming Multiprocessor** (24) 2) **shared memory limit per block** (48K), 3) **number of threads per block** (1024) and 4) **warp size** (32). The other parameters are less important because our tasks are unlikely to touch the limits of them.

Table 1: Blocking of Matrix Multiplication

Kernel				**A**		**B**		**C**		**Block**		Cache	#Thread
Size	In	Out	#MM	H	W	H	W	H	W	H	W	Usage	/ Block
1×1	512	128	1	14×14	512	512	128	14×14	128	49	1	43K	512
3×3	128	128	36	16	128	128	128	16	128	2	1	40K	1024
1×1	128	512	1	14×14	128	128	512	14×14	512	49	4	37K	512
1×1	1024	256	1	14×14	1024	1024	256	14×14	256	49	1	38K	1024
3×3	256	256	36	16	256	256	256	16	256	2	1	48K	1024
1×1	256	1024	1	14×14	256	256	256	14×14	1024	49	4	42K	1024

All the matrix blocking details are listed in Table 1. We will take the second line in it as an example to show how we use the resources. In this case, we will apply 128 different $3 \times 3 \times 128$ convolution kernels on a input $16 \times 16 \times 128$ (after padding) to get a feature map with size $3 \times 3 \times 128$. Since we are using Winograd Algorithm, we will form 36 matrix multiplications as $C = AB$ and their size are listed in the table. We decide to split each MM into two parts along the first dimension so that we will get two 8×128 matrices as output independently. Thus, there are 72 thread blocks in total, which is a multiple of the number of SM, so that all the SMs may be used evenly in this computation. The thread number is large and also a multiple of the warp size. The floating point numbers loaded into the shared memory for a single block is (8×128) of A and (64×128) of B and (8×128) of C, which is 40K in total and satisfy the constraint. The reason why we load (64×128) floating point numbers of B into shared memory is that, since B cannot fit into the cache, we have to deal with it iteratively, read and compute. However, frequent alternation between reading and computing among a bunch of threads is really harmful to the performance. So, we want the threads to read as much data as possible and do as much computation as possible in order to reduce alternation and increase the utilization of shared memory. We also pay great attention to the thread coalescence of our program. As we can see, the amount of data loaded into the shared memory and the total number of operations are multiples of the number of threads so that adjacent threads will access adjacent memory and do similar work.

The reason why our kernels can reach better performance than cuDNN is four-fold. First, we used the correct algorithm (Winograd and MM) and implementation strategy (outer-product, balanced thread workload, high shared memory utilization). Second, we take the kernel transformation step (by matrix G) in Winograd algorithm offline, because we are doing inference and the model is already given. Third, we fully tune the parameters for this given model on a single device. Fourth, we combine convolution, BN and activation into the same kernel without re-accessing the global memory. In addition, we also summarize several **takeaways of CUDA programming**:

- Minimize global memory access / Maximize the utilization of shared memory
- Maximize coalescence (e.g.threads in a warp should access adjacent memory address)
- Avoid branches/workload imbalance (especially within each warp)
- Maximize Occupancy (Shared Memory, #Registers, #Threads)

5 Experiments

In order to check the correctness of our kernel implementation, we compare the computation results with those computed from cuDNN. The input data and kernel weights are generated randomly. For all the elements in the output tensor, we only claim our kernels are correct if the maximum absolute difference is at magnitude 10^{-5} and the percentage of elements with maximum absolute difference larger than 10^{-5} is less than 0.1%. The running time of each kernel is computed by taking the average of 100 experiments on a completely idle GPU device. Note that we only compute the time of the actual computation (convolution, BN, activation) rather

Table 2: 3×3 Kernels

Kernals	Operations	128 / 128	256 / 256
cuDNN	MM + BN + ReLU	214us	384us
cuDNN	Winograd + BN + ReLU	95us	155us
Our Kernel	Winograd + BN + ReLU	**59**us	**117**us

Table 3: 1×1 Kernels

Kernals	512 / 128	128 / 512	1024 / 256	256 / 1024
Operations	MM+BN+ReLU	MM+BN	MM+BN+ReLU	MM+BN
cuDNN	117us	115us	219us	214us
Our Kernel	**58**us	**55**us	**186**us	**181**us

than the memory copy or any other overhead from data preparation, so that our comparison is fair. Also note that the operations we conduct after each convolution computation are the ones used in ResNet.

The results in Table 2 shows the experiment results of 3×3 kernels. Since there is also a winograd algorithm implementation in cuDNN, we compared our kernels with both of their implementation. We can see that winograd algorithm can truly boost the performance, and our implementation can be even more stronger. We get 39% and 25% performance improvement on kernels with 128 channels and 256 channels, respectively. Similar results for 1×1 kernels are listed in Table 3. Although winograd algorithm cannot be used on 1×1 kernels, our implementation using only general MM can still get 50% and 18% performance improvement on kernels with 128 channels and 256 channels, respectively.

6 Conclusion

There are a lot of tricks in parallel programming we can explore when trying to get our performance close to or even better than cuDNN. We can say that, by implementing convolution kernels with better performance, we have already unveiled the core technique of doing fast convolution computation on GPU, although there are still a long way to go to have a library as robust and comprehensive as cuDNN. In terms of the future work, develop a model-specific inference framework, instead of using the existing ones, should be the best way to deploy DNN models with high efficiency into practice.

References

[IS15] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International Conference on Machine Learning*. 2015, pp. 448–456.

[He+16] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[LG16] Andrew Lavin and Scott Gray. "Fast algorithms for convolutional neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4013–4021.

[NVI] NVIDIA. *The NVIDIA CUDA Deep Neural Network library (cuDNN)*. `http://https://developer.nvidia.com/cudnn/`.