5-2017

Optimization of Spatial Convolution in ConvNets on Intel KNL

Sangamesh Nagashattappa Ragate
University of Tennessee, Knoxville, sragate@vols.utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Sangamesh Nagashattappa Ragate entitled "Optimization of Spatial Convolution in ConvNets on Intel KNL." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Jack Dongarra, Major Professor

We have read this thesis and recommend its acceptance:

Stanimire Tomov, Hairong Qi

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Optimization of Spatial Convolution in ConvNets on Intel KNL

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Sangamesh Nagashattappa Ragate

May 2017

I dedicate this thesis to my parents N.S.Ragate and Manjula...

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Jack Dongarra for providing me this wonderful opportunity to work as a graduate research assistant at The Innovative Computing Laboratory(ICL). I would like to thank my supervisor Heike Jagode for the continuous support and guidance throughout my graduate studies. Also, I would like to thank Dr. Anthony Danalis and Dr. Azzam Haidar for being excellent mentors for my research at ICL.

I would like to thank all my teachers who taught the courses I took during my masters curriculum. I thank Colfax-Research team for providing me an opportunity to work as an intern which greatly influenced the content of my work presented in this thesis.

Last but not the least, I would like to thank my friend David Eberius for those lengthy discussions we used to have in the name of problem solving and other friends Hanumanth, Thananon Patinyasakdikul and Reazul Hoque for the support and the good times.

Appreciate your parents. You never know what sacrifices they went through for you...

Abstract

Most of the experts admit that the true behavior of the neural network is hard to predict. It is quite impossible to deterministically prove the working of the neural network as the architecture gets bigger, yet, it is observed that it is possible to apply a well engineered network to solve one of the most abstract problems like image recognition with substantial accuracy. It requires enormous amount of training of a considerably big and complex neural network to understand its behavior and iteratively improve its accuracy in solving a certain problem. Deep Neural Networks, which are fairly popular nowadays deal with such complicated and computationally intensive neural networks and their training process usually takes hours,days or, in some cases months,to achieve a particular accuracy. This opens up an opportunity for code modernizers and computer architecture experts to systematically study and optimize the core computational modules of the deep neural networks or simply DNN's.

In this thesis one such module called Spatial Convolution module is selected. This module is seen in most popular DNN architectures today and is also a major computational bottleneck. It is optimized for Intel Architecture, specifically the Knights Landing Architecture which is a powerful many-core processor introduced recently by Intel. The main strategy here was to use an unconventional convolution algorithm called Winograd Convolution algorithm to decrease the number of arithmetic operations. However, the reduction of arithmetic operations in Winograd algorithm comes at the cost of complicating the memory accesses. So the objective was to use

some advanced code optimization techniques to make the Winograd convolution as efficient as possible leading to fast spatial convolution in DNN.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditionally, the field of High Performance Computing was of interest only to research groups working exclusively on scientific computing or other areas that required the use of HPC systems. That has changed substantially in the recent times since the HPC systems are playing a key role in the advancement of areas like big data and machine learning applications that influence our day to day activities, [Coates et al. (2013)].

Deep learning, a sub category of machine learning is of special interest because of its ability to extract features automatically from the problem which makes it best fit for applications like image recognition, natural language processing ,etc., where it is very difficult to extract the important features that can be generalized. Deep learning is considered, by far the closest, to the way living organisms perceive or process image data. There are considerable number of research groups that are making very good progress in the theoretical concepts of deep learning, but, it still requires someone to focus on the implementation part on the silicon for its evaluation and understanding. NVIDIAs GPUs and Intel's Knights Landing architectures, which were earlier in use for scientific computing and in some graphical or visual applications, are now finding their ways into these highly acclaimed areas.

Optimization of the core modules of DNN will help speed up its analysis and evaluation. The main focus in the industries in recent times has been to develop fast, efficient and scalable deep learning frameworks [Dean et al. (2012)] and libraries. The masters thesis thus presented will talk about the FALCON library [Ragate et al. (2016)] which is the fast parallel library for one of the most compute intense module called the spatial convolution module found in many popular deep learning architectures. In this chapter, one such popular DNN architecture called VGG-16 D from the Visual Geometry Group of the University of Oxford will be introduced and later the same network will be used for performance benchmarks.

1.1 Motivation

Neural networks and the backpropagation technique to train them, have been around for quite some time now. The DNN architectures like convolutional neural networks that show excellent results in the Imagenet competitions and other image analysis applications have the same underlying mechanisms as the neural networks before. With very little tweaks and minor modifications, they produce the accuracies in image recognition that is comparable to a human and sometimes better than human. So the question we need to ask is "What are the factors that led to these amazing results ?". The factors are, the increased size of the networks, the huge data set that is used for training and the HPC technologies that allow these networks to run seamlessly, reducing the development and evaluation time. Even the slightest improvement in the performance of a fundamental block in the DNN architecture provides asymptotic improvement in the overall timing of the networks. This serves as the key motivation for the work of optimizing the spatial convolution module of a convolutional neural network for specific processor architecture, in this case the Knights landing architecture of Intel.

1.2 Convolutional Neural Networks

The fundamental modules present in convnets designed by VGG [Simonyan and Zisserman (2014)] are spatial convolution module, max-pooling module, ReLU(Rectified Linear Unit) and fully connected module. Initial layers of a convnet consists of 1-3 spatial convolution modules or simply conv modules followed by max-polling module and a ReLU. The final layers of a convnet consists of only fully connected neural networks. The layers with conv modules and fully connected modules are called as weight layers. Thus, the name VGG-16 means that there are a total of 16 weight layers in the convnet.

Table 1.1 shows the different VGG convnet configurations. The convolution modules are represented by "conv3" where 3 signifies the filter dimension, which is 3x3. Small filters of dimensions 3x3, 2x2 and even 1x1, as seen in VGG-16 C network, are widely used instead of bigger filters, in most of the recent and advanced convnet configurations. Since the current thesis is concerned only about optimizing the networks for speed and efficiency, it will not discuss the design concepts and the actual science that goes behind these networks. The networks are studied only from the implementational or engineering perspective. Out of these networks we arbitrarily select VGG-16 D network as the benchmark to understand the performance of the optimization strategies we discuss in the later chapters.

VGG convnets were designed for image recognition applications. As a result the input to the network will be an image of dimension 224x224 with 3 channels(red,blue and green). As the image is processed through the layers of the convnet, the height and width of the image starts decreasing due to the max-pool modules which perform sub-sampling of the output from the conv module and the number of channels starts increasing that is directly related to the number of filters used for convolution. As seen in the network, the number of filters of dimension 3x3 increases from 64 to 512 by the end of the network. Soft-max is a loss function that computes the error between the output of the network and the ground truth. This information is then passed in the

backward flow to perform gradient descent and update the weights, which in short, is the training process. The sections that follow try to give a gentle introduction of these fundamental modules of VGG. In the next and upcoming chapters, only convolution module will be studied in detail since it is the most dominant of all and also the most time consuming module.

Table 1.1: VGG ConvNet Configurations

A	B	C	D	E
11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
Input Image (224x224 RGB)				
conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool				
conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool				
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool				
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool				
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool				
FC-4096				
FC-4096				
FC-4096				
soft-max				

1.2.1 Fully Connected Module

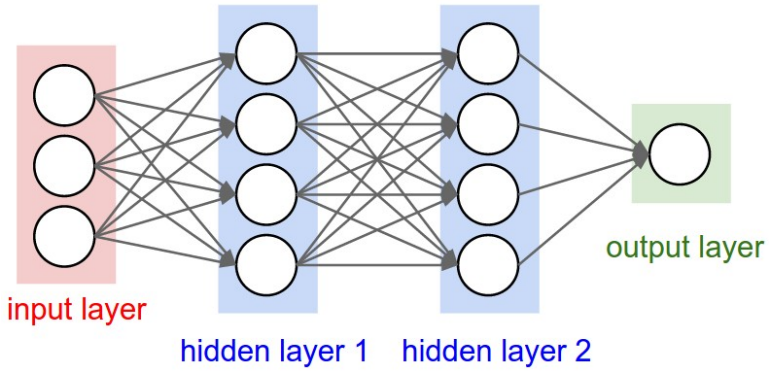


Figure 1.1: Fully Connected module.

Figure 1.1 shows a simple network made up of only fully connected modules or layers named as hidden layer 1 and hidden layer 2. The key aspect in the fully connected module is that every input is connected to every neuron symmetrically. The output of each neuron is weighted sum of the inputs plus the bias followed by an activation function. The number of output values coming out of each layer is equal to the number of neurons in that layer. The weights corresponding to each neuron in a particular layer can be viewed as a matrix W of dimension MxN where M is the number of neurons in the layer and N is the number of inputs coming to that layer, represented as a vector I. The output array Y can thus be computed using simple matrix vector product as Y=WI, which is of complexity O(MN).

1.2.2 Spatial convolution Module

Figure 1.2 shows convolution operation. In convnets, the input at each layer is arranged in a 3D tensor and the output of the convolution is also a 3D tensor. The depth of the input tensor should be equal to the depth of the filter but the height and weight of the filter are usually much smaller and in our case, they are fixed to 3. The 3D tensor of the filter is also known as the receptive field. The dot product of the filter and the volume of input that falls in its receptive field produce the output of a

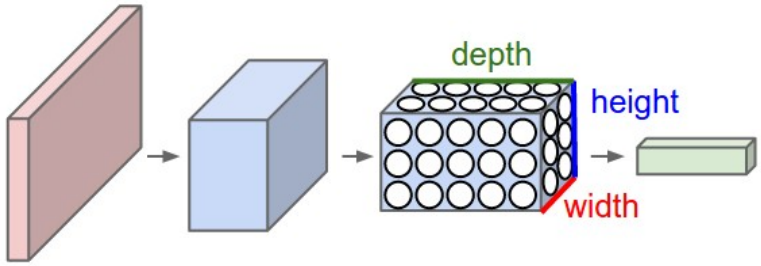


Figure 1.2: Convolution module.

neuron. Convolution in a particular layer involves convolving the input with multiple filters. One such filter traverses the input 3D tensor to produce one 2D output. These 2D outputs from multiple filters form the 3D output that is passed to the next layer. As the filter traverses across the input, there is an overlap of successive receptive fields both vertically and horizontally that decides the output height and width, the details of which will be covered in chapter 2.

1.2.3 Max-pooling Module

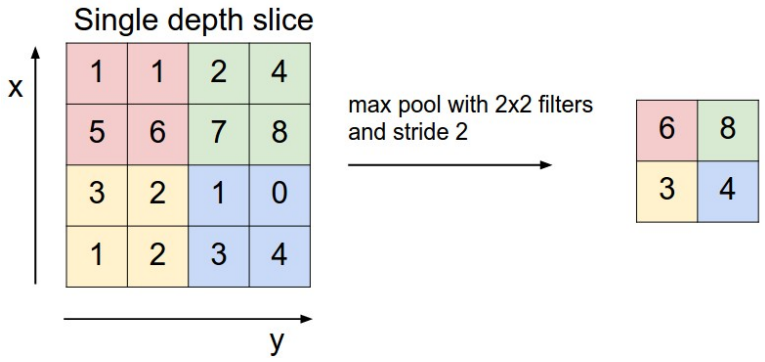


Figure 1.3: Max-pooling module.

Figure 1.3 is quite clear in describing the working of max-pooling layer. In the VGG configuration only 2x2 max-pooling is used in which the input is divided into non-overlapping 2x2 tiles and the output of that tile is either max of all the values in it or other math operation on the 2x2 values. Input is a 2D plane and the output is

also a 2D plane in which the height and width are scaled by half. This is also called down-sampling and provides spatial invariance capability to the network.

1.3 Putting it all together

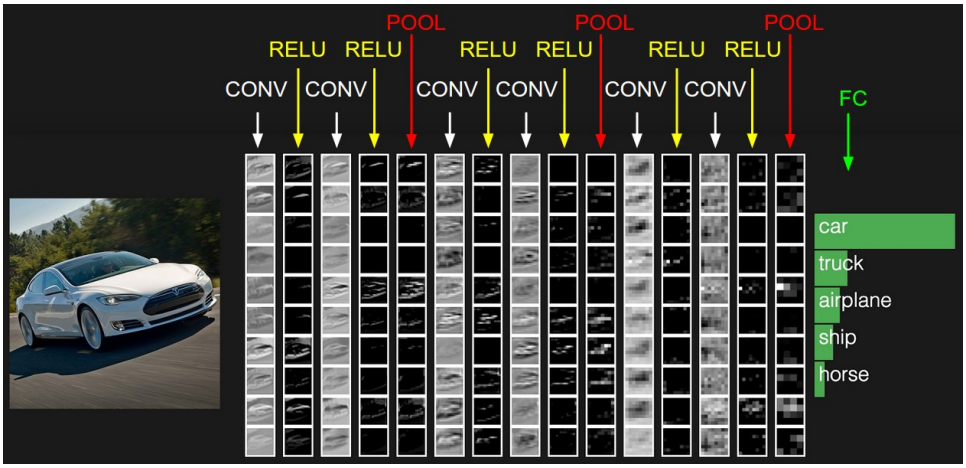


Figure 1.4: Image recognition using ConvNet.

Figure 1.4 (thanks to Karpathy's cs231n blog, Karpathy (2016)) shows the working convnet that is used in image recognition application. The image in the form of pixel data is processed one layer at a time to produce the final output which basically is a decision output. The layers of the convnet extract the features of the images automatically, which is the true nature of any deep learning system, and then they learn to classify them based on the training process. The working of a convnet is a simplified version of the process that happens in our visual cortex, to process image data, but still effective enough to produce very good recognition or classification accuracy.

Chapter 2

Spatial Convolution

2.1 Intuition

Spatial convolution is an important module of a convnet that does most of the work and it is computationally intensive. Typically, convolution in a particular layer consists of a set of F filters whose weights or coefficients can be trained to learn a random feature present in the input. In convnets, the height R and the width S of the filter represents the receptive field of the filter, which is usually much smaller than the height H and the width W of the input. However, the depth of the filter C, is the same as the depth of the input volume it is processing. As a result, the output of neuron is the dot product of the weights of the filter and the corresponding local receptive field it involves in the input. The total number of weights in a filter are therefore RxSxC and there are F filters. The weights of a filter are spatially invariant and they traverse across the width W and height H of the input volume to generate a 2D output plane of neurons, also called activation map. The 2D outputs from F such filters are stacked to form the overall output of the spatial convolution.

Each neuron sees only a small portion of the input but there is an overlap among the neighboring neurons both horizontally and vertically, which helps in information flow among the neurons. The local connectivity also makes it computationally efficient

in processing the image data since it is not practical for each neuron to exchange information with every other neuron. The filters in the beginning layers learn to recognize simple fundamental features of the image, like line segments of a particular orientation, an irregular patch of a specific color etc. The filters in the later layers use this information as building blocks to reconstruct more complex features like some regular structural patterns, wheel patterns etc. Also, the reason to have multiple filters in one layer is to learn as many as possible features from one particular input volume.

The output dimension O of the 2D activation map generated by one filter is dictated by the zero padding P used around the input volume, the extent of overlapping across the neighboring neurons D and the filter dimension R (usually R=S). It is described in the equation 2.1. The main requirement is that the parameters P, R and D should be selected such that the output dimension is an integer, which means that the number of neurons computing the activation map should be a positive integer.

$$O = 1 + \frac{(W - R + 2P)}{D} \tag{2.1}$$

2.2 Existing Implementation and Optimizations

This thesis is only concerned about optimizing the convnet architecture, specifically spatial convolution for Intel architecture. The Intel KNL architecture with 64/68/72 cores equipped with powerful 512-bit vector units is well suited for high performance computing. Direct convolution involves irregular memory accesses and since the filter size in VGG convnets are very small, it is difficult to vectorize the direct convolution operation which results in its very poor performance. An alternate technique widely used is the unfold and GEMM technique, discussed in the paper Chetlur et al. (2014) . The first task of this thesis was to understand the performance results of the existing implementations and try some code optimization techniques on them.

For this reason, a popular deep learning framework called Torch was selected and the Spatial convolution module in it was studied in detail. The current implementation as of today, involves unfold operation followed by CBLAS GEMM operation. The unfold technique reduces the input data which is in the form of 3D tensor to a 2D matrix. The weight data, which is a 3D tensor can be directly interpreted as a 2D matrix. The details of the unfold technique can be found in the paper Chetlur et al. (2014). The output of GEMM is consistent with the output of direct convolution and hence can be passed to the subsequent layers for further processing. A VGG-16 convolutional network was implemented using the torch framework on an Intel KNL processor configured in cache mode and a batched GEMM function was implemented to achieve high performance for the GEMM operation. This was done to understand the timing of the forward pass of the network and it was observed that 70% of the total time, 1.4s of 2s (MKL SGEMM timing), was only due to the spatial convolution operations.The unfold operations involve memcpy or streaming copy functions and the FOR loops were made parallel using openMP threads. There was nothing much to improve in the unfold operation and hence only the optimization techniques for GEMM will be discussed below with respect to a benchmark that involves only convolution operations of the VGG-16 D network.

2.3 Gemm for spatial Convolution

VGG-16 D network consists of 16 layers. There are in total 13 convolution operations and the corresponding weight and input matrices obtained for those layers after applying unfold technique are shown in the table 2.1 below. The max-pooling, ReLU and the fully connected layers which are not so computationally intense are not considered in the performance benchmarks. The input to the VGG-16 net is a 224x224 image data with 3 channels i.e red, blue and green. Hence the input tensor to the vgg-16 net is of dimension 3x224x224. The implementation considers single precision floating point data and operations only. First, the performance of the GEMM for

Table 2.1: VGG-16 D Convolution modules & corresponding Matrices.

Conv	Weight Matrix	Data Matrix
1	64x27	27x50176
2	64x576	576x50176
3	128x576	576x12544
4	128x1152	1152x12544
5-7	256x2304	2304x3136
8	512x2304	2304x784
9-10	512x4608	4608x784
10-13	512x4608	4608x196

these layers will be discussed based on different implementation techniques then the over all timing of unfold and GEMM will be presented for future comparisons.

2.3.1 MKL SGEMM

The first method to implement GEMM was to directly use the Intel's Math Kernel Library functions for SGEMM instead of CBLAS. The batch size consists of small subset of total images that are used together to train the network. For batch size T, there are T input matrices and one weight matrix at each layer for GEMM operation. A batched SGEMM function was developed using openMP parallel constructs, launching omp threads equal to the batch size and each thread calling MKL SGEMM function. The SGEMM function operates on the corresponding input and weight matrices based on the batch index. The performance of this implementation was impressive for all matrix shapes of the VGG-16 D net except for the first two which were the major performance bottlenecks.

Figure 2.1 shows the performance of the batched SGEMM function implementation using MKL SGEMM function. The second layer convolution consists of matrix shape M=64 K=576 and N=50176. This is a completely skewed matrix shape and MKL SGEMM performance on this layer was very poor. After many runs on similar matrix shapes it was observed that MKL SGEMM needs a more regular matrix shape for high performance.

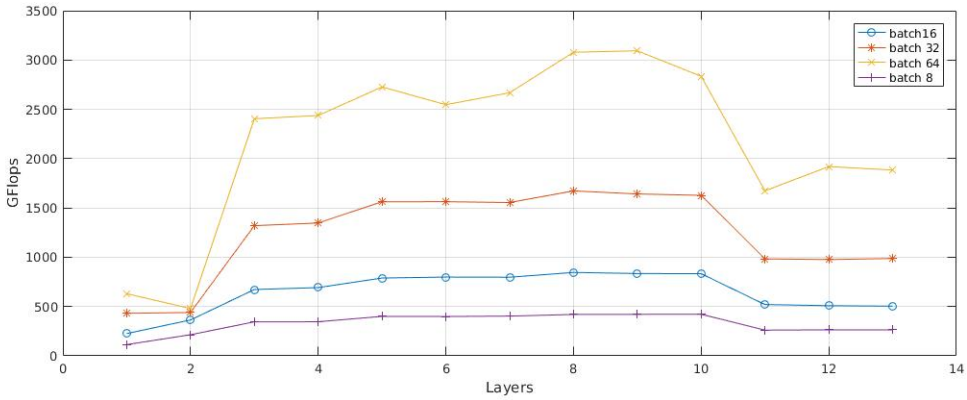


Figure 2.1: MKL SGEMM performance.

2.3.2 Batched SGEMM using code modernization techniques

An attempt was made to implement a batched SGEMM function using vectorization and other parallel computing techniques. The motivation was to try and beat or at least match the performance of the MKL SGEMM. OpenMP tasks were used to address the task level parallelism and AVX-512 VPU was used to address the data parallelism. The concept of recursive division was used to strategically split the matrices into tiles that can be processed in parallel as independent tasks. The recursive splitting was designed to traverse the matrix tiles in a systematic way that reduced the last level cache misses. At the base case of this recursion, tiling and register blocking techniques were used to effectively utilize the lower level caches with smaller sizes.

Only C programming with omp vectorization pragmas and ICC compiler optimization features were used in this implementation. It was observed that, for the tiling implemented in the base case multiply function, ICC was able to vectorize efficiently only if the complete tile of the product matrix was fetched to a temporary storage before the product computation and then written back to the original block after computation. This resulted in back to back generation of the fmadd instructions that leads to the best case usage of the fmadd pipeline. This is required because fmadd

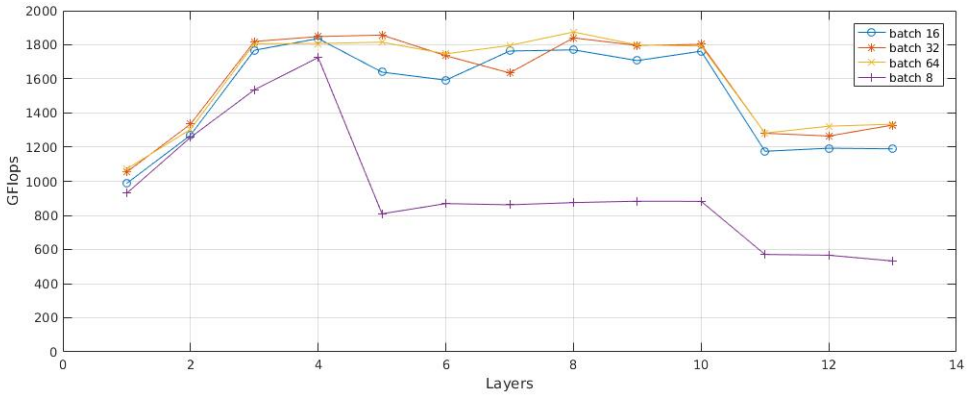


Figure 2.2: SGEMM performance.

instructions have considerable initial latency that can be covered to achieve higher throughput only by executing series of fmadds next to each other in every clock cycle.

The performance of this implementation on KNL was quite impressive even though it was a simple design. Figure 2.2 shows the performance plot and it can be seen that this implementation performs better than the MKL SGEMM for batch sizes lesser than 64, for all the matrices of VGG-16 D net. This implementation has higher performance than the MKL SGEMM on the first two matrix shapes for all the batch sizes.

2.3.3 RECURSIVE MKL

The third implementation uses the ideas of both the previous implementations. From the first implementation it was observed that MKL requires regular shaped matrices for achieving high performance and the second implementation gives the efficient way of dividing the matrices into smaller matrices. This recursive division can be controlled to output regular shaped smaller matrices favorable for MKL SGEMM. Hence, this implementation uses the recursive divide and conquer calling the MKL SGEMM at the base case.

The second GEMM of the VGG-16 D net consists of a bigger portion of the overall time and hence it is required to make this layer as fast as possible. Figure

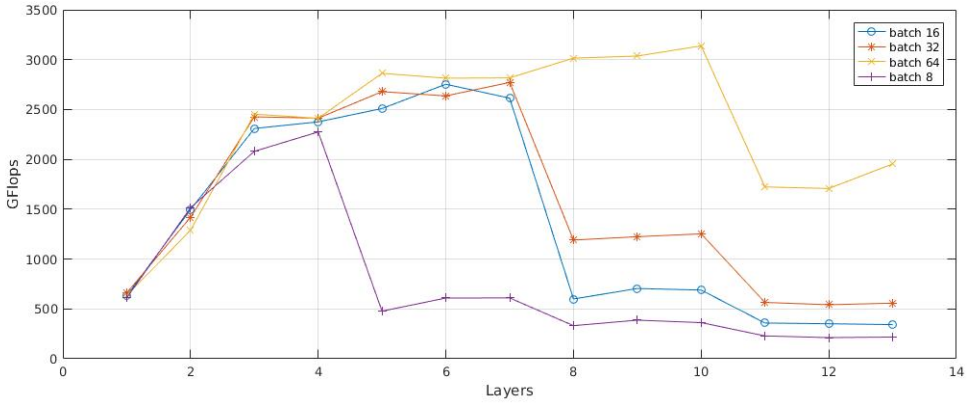


Figure 2.3: MKL REC performance.

2.3 shows the performance of this implementation and interestingly, the second layer GEMM achieves good performance with this implementation for all the batch sizes. Figure 2.4 shows the performance comparison plot for batch size 64 and it is quite clear that the third implementation achieves the best overall performance beating the previous implementations. Using the recursive division of the matrix, tiles of regular shapes were produced for the second layer GEMM and hence the MKL SGEMM function worked very well on the tiles leading to performance similar to the second implementation. The plot looks like an envelope touching the best performance points of both the implementations. The first GEMM of the VGG-16 D net does not contribute for much of the total timing since the number of weight filters and the input channels are very low. Hence, it is acceptable to have little less performance in this layer, provided, the other layers compensate by reaching high performance, leading to a very good overall timing.

2.4 Conclusion

Table 2.2 shows the total time taken for spatial convolution in VGG-16 D benchmark. The timing includes both unfold and GEMM operations. The best implementation takes 1.07 seconds to perform convolution operations in VGG-16 D net. The rest

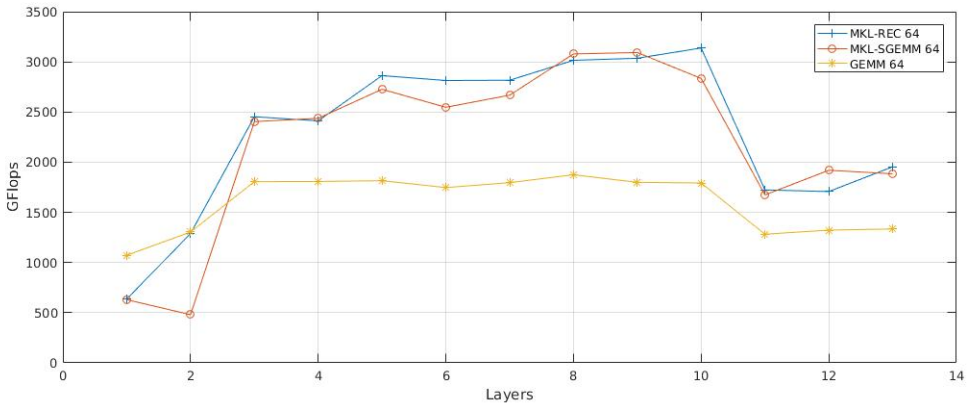


Figure 2.4: Performance comparison on VGG-16 D layers for batch size of 64.

Table 2.2: Overall Spatial Convolution timing.

Batch Size	Implementation 1	Implementation2	Implementation 3
8	0.62s	0.35s	0.34s
16	0.77s	0.55s	0.44s
32	0.8s	0.77s	0.61s
64	1.41s	1.44s	1.07s

of the operations like fully connected layers, max-pooling, ReLU and soft-max error computation together take around 0.6s roughly. Therefore, we can estimate forward path time or the inferencing time for 64 images to be 1.67s for the RECURSIVE MKL implementation. The main intention of this chapter was to introduce the spatial convolution operation conceptually. The preliminary optimization techniques discussed here give us a rough idea of the existing performance numbers which serve as a reference or a baseline for future optimization technique that we are going to introduce in the next chapter.

Chapter 3

Winograd minimal filtering

Direct convolution of a batch of N input images of dimension HxWxC with F filters of dimension RxSxC requires O(NFHWCRS) floating point operations. This is a very high asymptotic complexity and requires irregular strided memory accesses leading to poor cache performance. Also, since the filters are very small, it is very difficult to vectorize the convolution operation which results in a very low arithmetic intensity of the application. A better known way of performing filtering or convolution operation is to use the transformation method. In this method, the input and the filter are transformed using a linear operator to a different domain where the convolution operation gets transformed to a straight forward element wise product operation. This reduces the number of arithmetic operations required for convolution at the cost of transformation and inverse transformations.

One such popular method is the Fast Fourier Transform method. It is required to have input and the filter of similar dimensions in order to use this transformation. To perform convolution between a single 2D image and a 2D filter frame, this method requires O(RS log(WH)) operations, compared to O(RSWH) operations of direct convolution. It is obvious that this method requires fewer arithmetic operations asymptotically, but, note that the size of the filter should be large enough to take advantage of it. In contrast, convnets have very small filters that require considerable

amount of zero padding that introduces redundant computations making this method particularly inefficient for this application.

Winograd minimal filtering algorithm, introduced by Shmuel Winograd, is very well suited for convolution with small filters. In simple words, Winograd minimal filtering algorithm works on one image tile at a time. The tile size and filter size are comparably small. The main steps consist of transformation of image tile and filter, element wise product and inverse transformation of the product tile to obtain the output of convolution for that tile. This algorithm has a slightly reduced arithmetic complexity compared to direct method which makes a huge difference asymptotically, there by reducing the total time taken for spatial convolution in convnets. Minimal filtering algorithm has been around for quite a long time now, widely used in digital signal processing applications, but it has gained a lot of importance only recently because of the use of small filters in convnets.

3.1 Algorithm details

For one dimensional signals, to compute 2 output values using a 3 tap filter, conventional method does 6 multiplications and 4 additions. Alternatively, Winograd provided the following minimal algorithm [Winograd (1980)],

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1 + m2 + m3 \\ m2 - m3 - m4 \end{bmatrix} \tag{3.1}$$

where the output can be computed by the following expressions from 3.2 to 3.7. Note that in this method it is possible to precompute the factors that emerge from the filter co-efficients. By doing so, this method requires only 4 multiplications and 4 additions which is a good reduction of the number of operations for this scale.

$$
\begin{aligned}
m_1 &= (d_0 - d_2)g_0, && (3.2)\\
m_2 &= (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}, && (3.3)\\
m_3 &= (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}, && (3.4)\\
m_4 &= (d_1 - d_3)g_2, && (3.5)\\
F_0 &= m_1 + m_2 + m_3, && (3.6)\\
F_1 &= m_2 - m_3 - m_4. && (3.7)
\end{aligned}
$$

In the matrix form the minimal filtering algorithm can be represented as 3.8

$$Y = A^T[(B^T d) \odot (G^T g)] \tag{3.8}$$

where, d is the small input vector segment of 4 elements, g represents the filter coefficient vector of 3 elements and the symbols B and G represent the matrices that transform the input and filter correspondingly. A is the inverse transform matrix multiplied with the output of the element wise product.

$$
\begin{aligned}
B^T &= \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, && (3.9)\\
G &= \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}, && (3.10)\\
A^T &= \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}. && (3.11)
\end{aligned}
$$

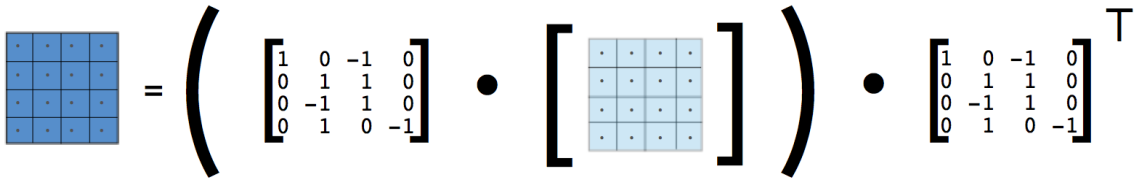


Figure 3.1: Input tile transformation.

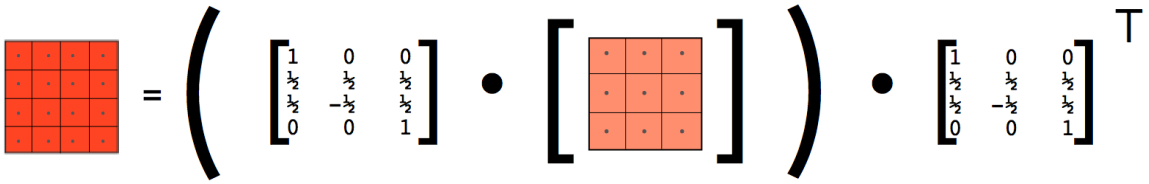


Figure 3.2: Filter transformation.

The same concept of minimal filtering can be extended to 2D image and filter data by modifying the matrix form of 3.8 to obtain 3.12.

$$Y = A^T[(B^T dB) \odot (G^T gG)]A \tag{3.12}$$

3.1.1 Case study

Let us apply the Winograd minimal filtering algorithm to a 2D input image filtered with a 3x3 filter. Let the tile size be 4x4 since we are dealing with the transform matrices 3.9 to 3.11 that are 4x4 for the input data, 4x3 for the filter data and 2x4 for the inverse transform that produces a 2x2 filtered output on one tile.

A 4x4 tile extracted from the input is transformed using the transformation matrix B as shown in the figure 3.1. Similarly, the 3x3 filter is transformed using the matrix G as shown in the figure 3.2.

Figures 3.3 and 3.4 show the element wise product operation of the transformed tiles and the inverse transformation to obtain a 2x2 filtered tile. In the conventional filtering, with filter stride equal to 1, a 3x3 filter operating on a 4x4 tile also generates the same 2x2 output tile as in figure 3.4. Such a Convolution operation is called

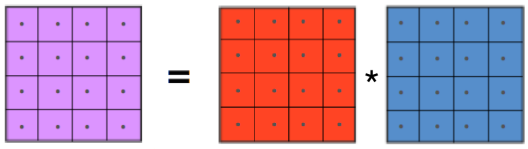


Figure 3.3: Element wise product.

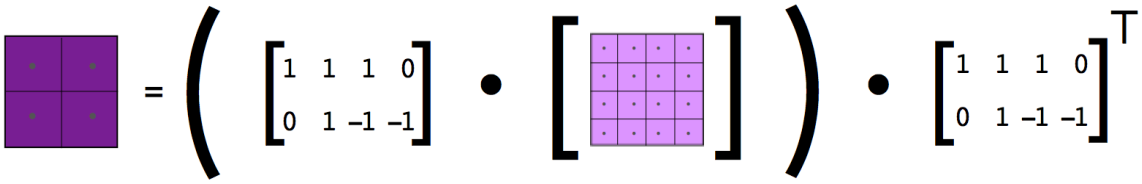


Figure 3.4: Output transformation.

perfect convolution because only those outputs are valid in which the receptive field of filter overlaps the input completely. This operation is repeated on all tiles of the 2D input image to form a corresponding 2D output frame whose height and width can be calculated using 2.1 mentioned in chapter 2. The total number of multiplications required in conventional filtering on one tile is 2x2x3x3=36 and using Winograd minimal filtering, ignoring the transformation, we need only 16 element wise multiplications. This results in an impressive speedup of $\frac{36}{16} = 2.25$ ideally, per tile of the image. The reason why we ignore the transformation cost becomes clear when we discuss the implementation of this algorithm in convnets.

Chapter 4

Implementation of Minimal filtering in Convnets

It is not trivial to obtain the ideal speed up mentioned in the chapter 3 by using the minimal filtering algorithm for spatial convolution. The steps of input transformation, element wise product and then output transformation on a tile by tile basis is an expensive process. In convnets, spatial convolution in any layer involves a total of NxC independent image frames of dimensions HxW each, where N is batch size and C is number of channels and a total of FxC filters of dimensions 3x3, 2x2 or even 1x1, where F is the number of filters. If the transformed image tile can be reused with the corresponding F filter tiles and if a transformed filter tile can be reused with the corresponding N image tiles, the transformation cost can be amortized for large values of N and F. This also means that the total number of arithmetic operations required is lesser than the direct convolution. To achieve this, it is required to transform the minimal filtering algorithm's element wise product to a GEMM operation.

4.1 GEMM in minimal filtering

In convnets, spatial convolution is performed between N images of size HxW and F filters of size RxS with C channels in both image and filter. Each image frame can have a total of T=(H-2)x(W-2)/4 tiles in it and there are NxC such frames. Minimal filtering for any tile can be written as shown in 4.1, where t is tile index, c is channel index, n and f are batch and filter indices respectively.

$$Y_{n,t,c,f} = A^T[(B^T d_{n,t,c} B) \odot (G^T g_{c,f} G)]A \tag{4.1}$$

Assuming that we have many image tiles and multiple filters, we can precompute transformed image tiles $U_{n,t,c} \equiv \left(B^T d_{n,t,c} B\right)$ and transformed filters $V_{c,f} \equiv \left(G^T g_{c,f} G\right)$. Also, it is to be noted that the dot products of corresponding tiles across the channels should be accumulated to perform convolution in a 3D volume. Hence, minimal filtering in convnets can be written as shown in the expression 4.2. This expression can be further simplified by moving the inverse transform outside the summation across C channels. This is allowed because the transform and inverse transform operators are linear and exploring this fact makes the computation efficient as shown in 4.3.

$$Y_{t,f} \equiv \sum_c Y_{n,t,f,c} = \sum_c A^T \left[U_{n,t,c} \odot V_{c,f}\right] A. \tag{4.2}$$

$$Y_{n,t,f} = A^T \left[\sum_c U_{n,t,c} \odot V_{c,f}\right] A \tag{4.3}$$

As discussed earlier, a transformed input tile should be reused to multiply with corresponding F filter tiles. Similarly, a transformed filter tile should be reused to multiply with corresponding input tiles across all the batches, N and as stated earlier, the element wise products of tiles across C channels are accumulated into one output tile. With this in mind, careful observation of 4.3 implies that, this can be turned into GEMM form by scattering the 16 elements of every tile to 16 different matrices

to form the inputs for the GEMM. The corresponding elements across the C channels are stored in a row and the corresponding elements from all the T tiles are stored in a column. Thus each 3D input image tensor of dimension HxWxC is converted to 16 matrices each of T=(H-2)x(W-2)/4 rows and C columns. Similarly, elements of transformed filter tiles are scattered to form 16 matrices each of dimension C rows and F columns.

Putting it all together, there are 16 matrices of dimension TxC for an image input and 16 matrices of dimension CxF that represent all the F filters, resulting in 16 one to one matrix multiplications. Also,the batch size is N which means that there are N such image matrices that have to be multiplied with the common filter matrices. The expression 4.4 shows the mathematical representation of the GEMM form. Turning the filtering algorithm into GEMM results in the maximum reuse of every transformed tile and increases the code-vectorization capabilities that will lead to better performance especially in KNL architecture which has powerful 512-bit vector units. In addition to this, there are many high performing BLAS libraries that provide easy to use GEMM APIs that makes the implementation simpler and more efficient. The whole process of reducing the minimal filtering algorithm to GEMM is very well described in [Lavin (2015)].

$$P^{x,y}_{n,t,f} = U^{x,y}_{n,t,c} V^{x,y}_{c,f} \tag{4.4}$$

In summary, convolution of 3D image tensor with a 3D filter tensor involves 16 one to one matrix multiplications, between the transformed image and filter matrices. The respective matrices that need to be multiplied are identified by (x, y) index. Finally, the output tile is restored by gathering the corresponding elements from the 16 product matrices. Inverse transform is then applied to all the restored tiles to obtain the final convolution output as shown in 4.5.

$$Y_{n,t,f} = A^T P_{n,t,f} A. \tag{4.5}$$

Algorithm 1 shows the pseudo-code for the implementation of minimal filtering and appendix A has the C code implementation of this. OpenMP is used for multithreading since it is more scalable and portable than pthreads. The ICC compiler pragmas are used for vectorization and loop unrolling wherever necessary.

Procedure input and filter transforms scatter the elements of the transformed tiles to form the matrices. Procedure GEMM performs the matrix multiplications and the procedure output transform gathers the elements of tiles and performs inverse transform on each tile.

4.2 Input Format

Input image format to the convolution module plays a significant role in achieving good performance results. The parameter "M" also called the merge parameter represents the input format used and also influences the shape of the matrices generated from the input images. Table 4.1 shows the cases that arise from different values of M.

Table 4.1: Merge parameter and the data format

M	Input Format	Output Format
1	NCHW	NF(H-2)(W-2)
N	CNHW	FN(H-2)(W-2)
$1 < M < N$	$\frac{N}{M}$CMHW	$\frac{N}{M}$FM(H-2)(W-2)

Input format NCHW means that the width of the image is the inner most dimension with stride 1 and the N is the outermost dimension with the longest stride. Keeping HW fixed, M 2D image frames of the same channel are merged across the batch. This is done to adjust the matrix shape to get good performance in GEMM since some of the layers have tall skinny matrices.

Algorithm 1 Fast convolution of the form F(2x2,3x3)

$N \leftarrow$ batch size
$C \leftarrow$ image channels
$F \leftarrow$ filters
$T \leftarrow (H-2)(W-2)/4$, tiles per input channel
$d_{n,t,c} \in \mathbb{R}^{4\times4}$ is tile t in input channel c of batch n
$g_{c,f} \in \mathbb{R}^{3\times3}$ is filter channel c of filter f
$Y_{n,t,f} \in \mathbb{R}^{2\times2}$ is tile t in output channel f of batch n
B^T, G and A^T are input, filter and output transforms
Neighboring tiles overlap by 2 pixels

procedure INPUT TRANSFORM
    for $n \leftarrow 0, N$ **do**
        for $c \leftarrow 0, C$ **do**
            for $t \leftarrow 0, T$ **do**
                $u \leftarrow B^T d_{n,t,c} B \in \mathbb{R}^{4\times4}$
                Scatter $U^{x,y}_{n,t,c} \leftarrow u_{x,y}$
end procedure

procedure FILTER TRANSFORM
    for $c \leftarrow 0, C$ **do**
        for $f \leftarrow 0, F$ **do**
            $v \leftarrow G^T g_{c,f} G \in \mathbb{R}^{4\times4}$
            Scatter $V^{x,y}_{c,f} \leftarrow v_{x,y}$
end procedure

procedure GEMM
    for $x \leftarrow 0, 4$ **do**
        for $y \leftarrow 0, 4$ **do**
            for $n \leftarrow 0, N$ **do**
                $P^{x,y}_{n,t,f} \leftarrow U^{x,y}_{n,t,c} V^{x,y}_{c,f}$
end procedure

procedure OUTPUT TRANSFORM
    for $n \leftarrow 0, N$ **do**
        for $f \leftarrow 0, F$ **do**
            for $t \leftarrow 0, T$ **do**
                Gather $p_{x,y} \leftarrow P^{x,y}_{n,t,f} \in \mathbb{R}^{4\times4}$
                $Y_{n,t,f} \leftarrow A^T p_{x,y} A$
end procedure

Table 4.2: Merge parameter and Matrix shape

#	W	C	F	Before Merge			Merge	After Merge		
				m	k	n		m	k	n
1	226	3	64	12544	3	64	1	12544	3	64
2	226	64	64	12544	64	64	1	12544	64	64
3	114	64	128	3136	64	128	4	12544	64	128
4	114	128	128	3136	128	128	4	12544	128	128
5	58	128	256	784	128	256	8	6272	128	256
6	58	256	256	784	256	256	8	6272	256	256
7	58	256	256	784	256	256	8	6272	256	256
8	30	256	512	196	256	512	16	3136	256	512
9	30	512	512	196	512	512	16	3136	512	512
10	30	512	512	196	512	512	16	3136	512	512
11	16	512	512	49	512	512	16	784	512	512
12	16	512	512	49	512	512	16	784	512	512
13	16	512	512	49	512	512	16	784	512	512

If the input frame is of dimension HxW, the output frame after convolution becomes (H-2)x(W-2), since we are using 3x3 filter in VGG-16 D net. The input output relation was already explained in chapter 2 using the relation 2.1. In order to have the output frame size same as input's, we use 1 layer of padding across the input frames for all the convolution modules that we use for the benchmark. Input for the first module is an image of size 224x224x3 (image frame is 224x224 and there are 3 such frames or 3 channels) and the size of the batch N is 64. After padding, each input image frame becomes 226x226 and number of tiles present in one such frame is $T = \frac{(226-2)\times(226-2)}{4} = 12544$. Table 4.2 shows the shape of the matrices before and after merging. Chapter 2 showed that the MKL SGEMM performance was very low on tall skinny matrices and hence the merge parameter allows us to adjust the matrix shapes that results in near to peak performance of GEMM using MKL.

Chapter 5

Implementation Results and Evaluation

5.1 FALCON library

Fast Parallel Convolution library is the C implementation of Winograd minimal filtering using 3x3 filters. It is optimized for 2nd generation Xeon Phi's, code named Knights Landing architecture, but is portable across any other Intel Architecture with very less performance penalty. It is an open source code published in github with MIT license for reuse and further improvement. The three main functions of the library are the initialization function, convolution function and the cleanup function. The convolution function internally performs transformation, GEMM and inverse transformation. The two important optimizations performed were the bandwidth optimization and the GEMM optimization.

5.1.1 Bandwidth optimization

The transform and inverse transform routines were mainly bandwidth bound since they don't have any significant computations. MCDRAM plays a key role in achieving high bandwidth on KNL [Asai (2016)]. The best way to optimize the bandwidth was

to use a scratch memory to store the transformed data structure and reuse that for all the spatial convolution operations in the convnet. This requires to have a scratch pad memory big enough to hold the biggest data structure of all the convolution operations. This was possible for the benchmark of our interest, i.e VGG-16 D, and it is quite possible for many popular convnet architectures because the MCDRAM is 16GiB in size.

The function hbw_malloc() is used to allocate the scratch pad memory on MCDRAM and it is done in the initialization routine. The transform and inverse transform operate on one 2D frame of size HW, at a time, instead of hopping across the channels or the batch. This small stride accesses result in good cache performance. Other trivial optimization involves loop unrolling, reducing the matrix multiplication in transform and inverse transform to simplified scalar arithmetic operation with constants, adjusting the stride in scatter and gather operations to a factor other than the multiple of 4096 bytes to avoid cache associativity misses.

5.1.2 GEMM optimization

Chapter 2 showed that the MKL GEMM performance is very low for skewed sized matrices. The first GEMM of the convnet is tall skinny, in other words skewed as shown in table 4.2. Since, this GEMM is bandwidth bound, it was optimized by writing a 3 FOR loop GEMM in C with tiling and vectorization which gave better performance than MKL GEMM. The next and the most important optimization used was the merge parameter to adjust the shape of the matrices to achieve near to peak GEMM performance using MKL.

5.2 Performance

Results reported here are obtained on a 68-core Intel Xeon Phi processor 7250 with 96 GiB of DDR4 RAM and 16 GiB of MCDRAM in flat mode. The system was running

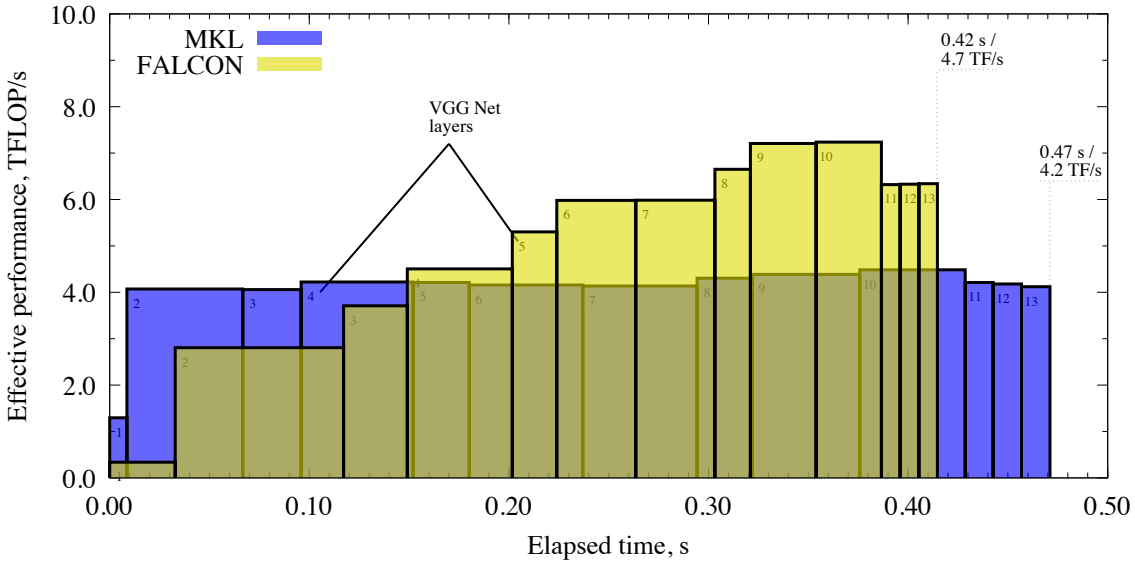


Figure 5.1: VGG-16 D convolution modules performance in FALCON and MKL.

CentOS 7.2 with stock kernel. The code was compiled with Intel C compiler 17.0.0.098 (Build 20160721) and linked with Intel MKL 2017 (build date 20160802). VGG-16 D was the selected convnet for the benchmark. The benchmark was designed to perform only the 13 convolution operations, since our optimization deals with only convolution, and also convolution is the most time consuming operation in VGG convnets. For comparison, we use Intel's MKL DNN primitives for performing convolutions on the same benchmark and evaluate our results.

The metric used to measure performance is called as "Effective performance". It is measured in TFLOPS/s and is computed using the expression 5.1 where τ_{tot} is the total wall time taken for computation.

$$P_{\text{eff}} = \frac{2NFCRS(H-2)(W-2)}{\tau_{\text{tot}}} \tag{5.1}$$

The table 5.1 and the figure 5.1 show the effective performance of MKL DNN primitives and FALCON library. In the table, W is the width of the square image input as it passes through different layers of the VGG-16 D net, C and F are respectively the channels and number of 3x3xC filters used for convolution. The

cost is the number of floating point operations required for convolution computed using the numerator of the equation 5.1. MKL DNN primitives perform better for the first three convolutions and then the FALCON catches up, and eventually performs better than MKL primitives for rest of the convolutions resulting in the better overall performance. There is a 10% improvement in the overall timing for FALCON compared to the MKL primitives.

Table 5.1: Convolution performance in MKL and FALCON.

Convolution					DNN in MKL		FALCON	
#	W	C	F	Cost, GFLOP	τ_{tot}, ms	P_{eff}, TFLOP/s	τ_{tot}, ms	P_{eff}, TFLOP/s
1	226	3	64	11.1	8.6	1.30	32.8	0.34
2	226	64	64	236.8	58.2	4.07	84.4	2.80
3	114	64	128	118.4	29.2	4.06	31.9	3.71
4	114	128	128	236.8	56.1	4.22	52.6	4.50
5	58	128	256	118.4	28.1	4.21	22.3	5.30
6	58	256	256	236.8	57.0	4.16	39.6	5.98
7	58	256	256	236.8	57.3	4.14	39.6	5.98
8	30	256	512	118.4	27.5	4.30	17.8	6.65
9	30	512	512	236.8	54.0	4.39	32.9	7.21
10	30	512	512	236.8	52.8	4.48	32.7	7.24
11	16	512	512	59.2	14.1	4.21	9.4	6.32
12	16	512	512	59.2	14.2	4.18	9.4	6.33
13	16	512	512	59.2	14.4	4.12	9.3	6.34
Net				1964	471	4.17	415	4.74

Interesting thing to note is that the peak performance for SGEMM using MKL on KNL processor is approximately 4.6 TFLOPS/s. The effective performance of all the 13 convolution modules using MKL primitives are less than this performance. However, the effective performance of the convolution modules starting from module 5 to 13 using FALCON are well above 4.6 TFLOPS/s, even though MKL SGEMM is

Table 5.2: Effective bandwidth and performance in MCDRAM .

Fast Convolution					FALCON in MCDRAM					
#	m	n	k	M	τ_{in}, ms	P_{in}, GB/s	τ_{out}, ms	P_{out}, GB/s	τ_{MM}, ms	P_{MM}, TFLOP/s
1	12544	64	3	1	1.3	149	14.1	292	17.4	0.28
2	12544	64	64	1	25.2	164	13.8	298	45.5	2.31
3	12544	128	64	4	6.3	166	6.9	296	18.7	2.81
4	12544	128	128	4	12.4	167	6.9	299	33.3	3.16
5	6272	256	128	8	3.2	163	3.5	290	15.6	3.38
6	6272	256	256	8	6.4	164	3.6	287	29.6	3.55
7	6272	256	256	8	6.3	166	3.5	290	29.7	3.54
8	3136	512	256	16	1.8	153	1.8	292	14.2	3.70
9	3136	512	512	16	3.7	149	1.7	296	27.4	3.84
10	3136	512	512	16	3.5	157	1.7	294	27.4	3.83
11	784	512	512	16	1.2	133	0.5	262	7.6	3.44
12	784	512	512	16	1.2	134	0.5	258	7.6	3.44
13	784	512	512	16	1.2	133	0.5	258	7.6	3.46

being used in the FALCON to perform the GEMMS internally. This is the whole point of using Winograd minimal filtering because it will reduce the number of arithmetic operation required for convolution. The performance numbers of module 5 through 13 which are greater than 4.6 TFLOPS/s prove this point.

Table 5.2 shows the detailed performance results of the internal functions of the FALCON library in MCDRAM mode. The size of the MCDRAM is 16GiB and it can supply streaming data to the processor with the peak bandwidth of 400GiB/s. The size of the MCDRAM is sufficient to store all the data structures of the VGG-16 D net including the scratchpad memory. The average bandwidth obtained for input and output transformations were about 37.5% and 70% of the peak, respectively. The main reasons for these were because of small tile size, the irregular or non streaming data accesses within the tiles and 16 scatter/gather operations per tile during transformations.

The performance of the SGEMM is quite impressive with an average about 3 TFLOPS/s, reaching close to 4TFLOPS/s for some modules. To understand the significance of MCDRAM, same experiment was repeated without allocating the data

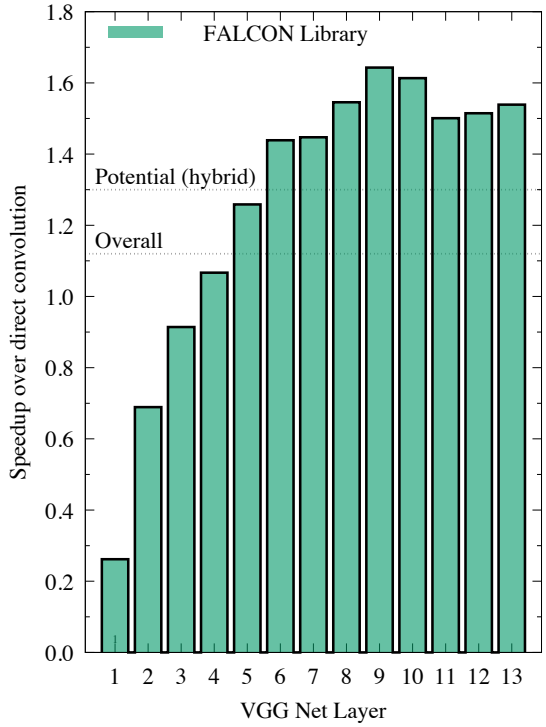


Figure 5.2: Speedup of FALCON over MKL .

Table 5.3: Effective bandwidth and performance in DDR4 .

Fast Convolution					FALCON in DDR4					
#	m	n	k	M	τ_{in}, ms	P_{in}, GB/s	τ_{out}, ms	P_{out}, GB/s	τ_{MM}, ms	P_{MM}, TFLOP/s
1	12544	64	3	1	4.1	47	60.2	68	85.4	0.06
2	12544	64	64	1	89.9	46	59.8	69	124.3	0.85
3	12544	128	64	4	22.4	46	30.3	68	51.4	1.02
4	12544	128	128	4	45.1	46	30.3	68	66.3	1.59
5	6272	256	128	8	11.4	46	15.0	69	27.3	1.93
6	6272	256	256	8	22.7	46	14.9	69	35.1	3.00
7	6272	256	256	8	22.8	46	14.9	69	35.1	3.00
8	3136	512	256	16	5.9	47	9.2	56	15.3	3.43
9	3136	512	512	16	11.8	47	7.5	69	29.9	3.52
10	3136	512	512	16	11.9	47	7.5	68	30.0	3.51
11	784	512	512	16	3.3	49	1.9	68	9.0	2.93
12	784	512	512	16	3.3	49	1.9	68	9.0	2.93
13	784	512	512	16	3.3	50	1.9	68	9.0	2.93

on MCDRAM. Table 5.3 shows the performance results with significant performance loss in transformations and the SGEMM.

Figure 5.2 shows the speed up achieved by FALCON for each module with respect to MKL DNN primitives which performs direct convolution. In chapter 3, it was shown that the ideal speed up of minimal filtering over direct convolution is 2.25 which is the upper bound and hence the speed up of all the modules are well below this value. FALCON achieves a maximum speedup of about 1.7 for module 9 and 1.1 overall speed up. MKL DNN primitives perform better for first 3 modules and the FALCON does well for the rest of the modules. As a result, there is a scope for a possible hybrid implementation which uses MKL DNN primitives and FALCON library in a complementing way to get the best of both. It can be predicted from the plot in 5.2 that such a hybrid can achieve a speed up of at least 1.3.

Chapter 6

Performance Analysis

In this chapter, a detailed performance analysis of the GEMM in first conv module is presented. This requires the use of PAPI (Performance API) tool from Innovative Computing Laboratory to tap into the hardware performance counters of KNL. PAPI provides a low level interface to access the hardware counters in the form of user friendly APIs.

6.1 Tall Skinny GEMM

In convnets, any technique like minimal filtering or unfold-GEMM that reduce the spatial convolution operation to a GEMM operation will suffer from forming a tall-skinny first layer GEMM that are not computationally efficient on MKL GEMM. In this design, one such GEMM is the first conv module gemm of dimensions,(m=12544,k=3,n=64). The theoretical arithmetic intensity achievable for such a GEMM in single precision is given by equation 6.1, and in this case it is 0.7327. In KNL, theoretical peak performance of SGEMM is 6 TFLOP/s, which requires an arithmetic intensity of 15, assuming 400 GiB peak bandwidth (BW_{peak}).

$$AI = \frac{2mnk}{4(mk + kn + 2mn)} \tag{6.1}$$

The theoretical peak performance for this SGEMM is $0.7327 \times BW_{peak}$= 293 GFLOP/s. The low arithmetic intensity makes this GEMM completely memory bound and also, since the matrix A which is of dimension 12544x3, can fit into L2 cache of KNL, it is suggested to have a streaming multiplication rather than blocking multiplication which is quite common in MKL.

Listing 6.1: C code for GEMM

```
void gemm_ker(int m, int n, int k, float* a, int lda,float* b, int ldb,
    float* c,int ldc){

    const int BLK = 16;
    int x, xx, y, z, i;
    for(z = 0; z < n; z++){
        for(x = 0; x < m; x += BLK){
            float p[BLK] __attribute__((aligned(64)));
            p[0:BLK] = 0.0f;
            #pragma unroll(3)
            for(y = 0; y < 3; y++){
                #pragma simd
                for(i = 0; i < BLK; i++){
                    p[i] += a[x+i+y*lda]*b[y+z*ldb];
                }
            }
            c[x+z*ldc:BLK] = p[0:BLK];
        }
    }
}
```

Matrix multiplication with cache blocking is very effective only if m,k and n sizes are comparable which results in maximum $\frac{flop}{byte}$ ratio. In a column major layout, if

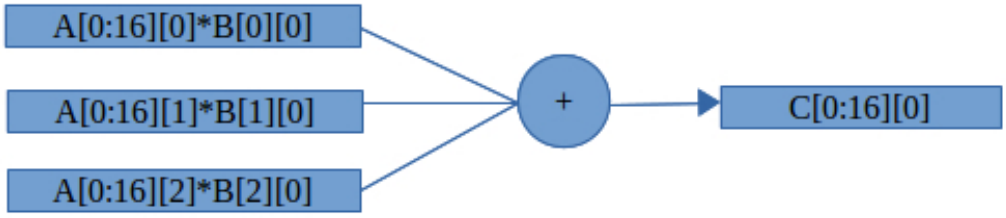
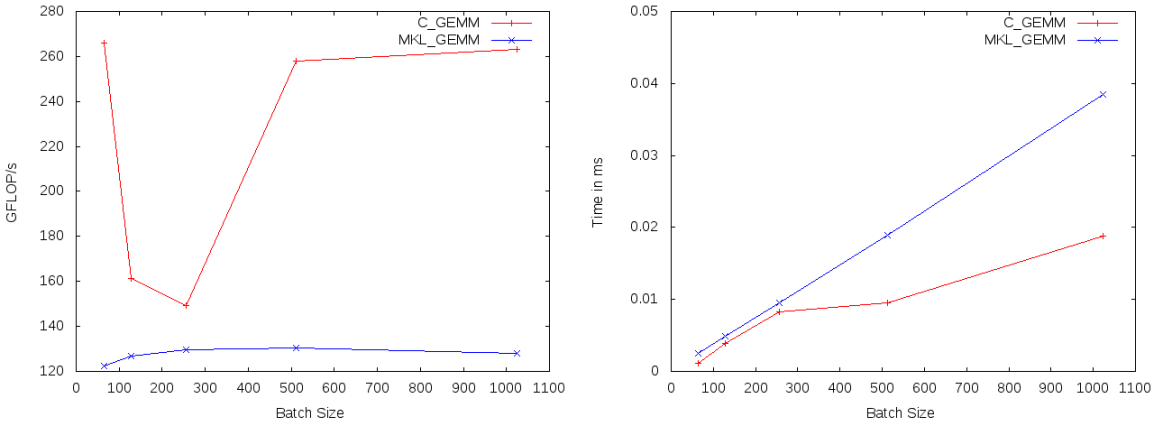


Figure 6.1: Streaming Multiplication.

$k << m$ and $k << n$, it is not required to have blocking for n. A simple C code implementation of GEMM is shown in the code 6.1.

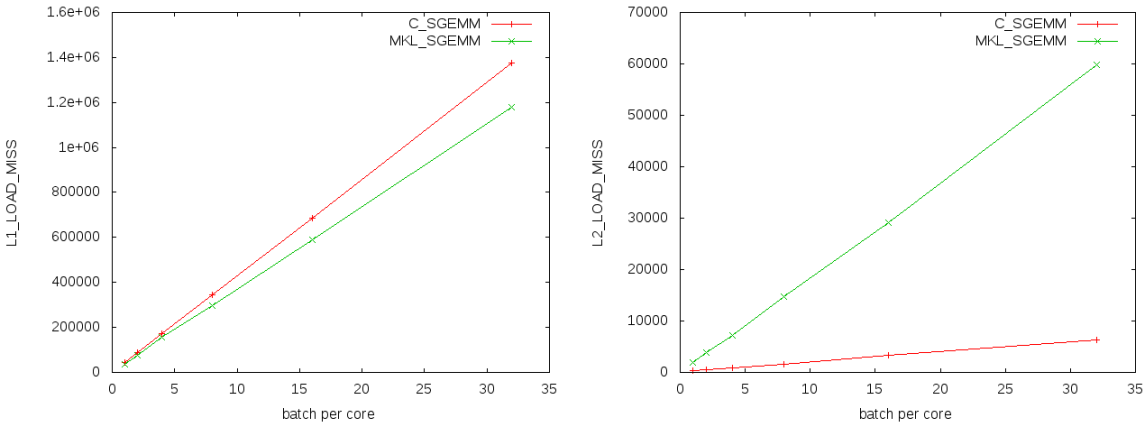
Figure 6.1 shows the the working of the C code that is optimized to have good bandwidth since the target GEMM is memory bound. The GEMM of interest is C=AB,(m=12544 ,k=3,n=64), where A in 12544x3, B is 3x64 and C is 12544x64. The layout is column major, with three input streams, unrolled along k but vectorized along m and one output stream. For each iteration, 3 vectors of A from 3 columns multiply with 3 corresponding elements of B in a particular column and accumulate to form one vector of C along the same column as B.



(a) FLOPS **(b)** TIME

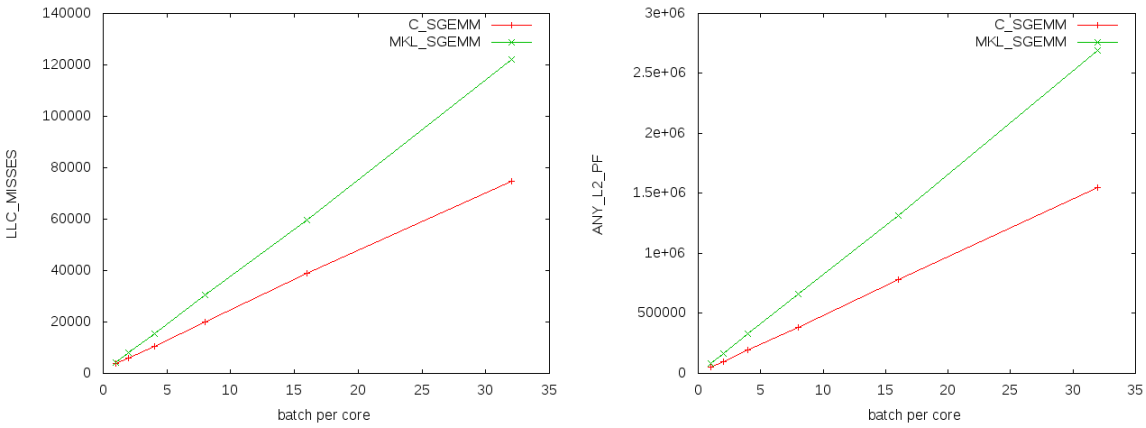
Figure 6.2: Performance plot of MKL GEMM and C implementation of GEMM

Figure 6.2 shows the flops and timing performance of the C SGEMM and the MKL SGEMM. C implementation with all its simplicity beats the MKL SGEMM for all the batch sizes. Specifically, batch size of 1024 is what was used in minimal filtering where the number of images in a mini batch were 64 but each one was reduced to 16 matrices leading to a batch size of 1024 for the GEMM.



(a) L1 MISS **(b)** L2 MISS

Figure 6.3: L1 L2 PAPI Performance counters



(a) LLC MISS **(b)** L2 PF

Figure 6.4: LLC L2-PF:PAPI Performance counters

Figures 6.3 and 6.4 give a more detailed insights on the performance of the GEMM with respect to the cache memory. In the actual implementation of the minimal

filtering algorithm, a batch size of 64, has 64 image-filter convolutions executed by 68 threads on 68 cores of KNL. Assuming, only 64 cores and one thread per core ,each thread works on one image-filter convolution which in turn has a batch of 16 matrix multiplications. The performance counters are therefore measured only for a single thread running on a single core to simplify the analysis.

The C code SGEMM implementation does not use any blocking and hence the L1 load misses are more than the MKL SGEMM which performs blocking by default. The input matrix A is small enough to fit in L2 cache and it is read 64 times to generate the complete C matrix. As a result, the C code implementation has very less L2 and LLC misses compared to MKL SGEMM. The only reason why MKL SGEMM seems to be having a higher L2 and LLC misses can be attributed to the unwanted blocking and prefetch processes. Since the GEMM of interest is completely memory bound, it is preferred to have streaming access of the input data rather than the blocking access. Figure 6.4 (b) shows that there is excess prefecthing in MKL SGEMM that might be replacing the required data. The actual code of MKL SGEMM is not open source so it is hard to comment on its behavior. Also, it is to be noted that, the implementation that gives good L2 and LLC performance at the cost of L1 performance is much better than the one that only gives good L1 performance because the higher level cache misses are very expensive in time.

In summary, reducing an algorithm to GEMM is only beneficial if the GEMM formed is in a good shape and not tall-skewed. MKL SGEMM in particular suffers for tall skinny GEMMS. It is required to identify such GEMMS and once identified, it is quite easy to write simple C code that achieves near peak performance for such shapes. In the current implementation, the peak performance of the tall skinny SGEMM was 293 GFLOP/s and the C code implementation was able to achieve 262 GFLOP/s where as MKL SGEMM did only about 128 GFLOP/s.

Chapter 7

Conclusion

This thesis presents the FALCON library, which implements fast convolution based on Winograd's algorithm with performance optimization for Intel Xeon Phi processors x200 (formerly Knights Landing).

7.1 Performance Optimization

Even though Winograd's minimal filtering algorithm reduces the number of floating-point operations necessary to compute convolution, it is not trivial to take advantage of these savings. Complex memory access pattern in input and output data transformations require carefully controlled data containers and memory access patterns as shown in FALCON source code. Performing matrix multiplication also required thorough tuning by fusing smaller matrices into bigger ones, adjusting the strategy of multi-threading, and injecting custom code in place of BLAS routines in special cases.

7.2 High-Level Language

Despite the complexity of code optimization, the FALCON code does not use any assembly or intrinsic functions for explicit access to platform-specific instructions.

Instead, it relies on automatic vectorization in the compiler, on standard functionality of the OpenMP framework, and on traditional BLAS routines. This simplifies future code maintenance and adaptation of the application to the upcoming computing platforms. Additionally, this case study proves by example the possibility of using high-level languages and frameworks in computational applications for Intel Xeon Phi processors.

7.3 Speedup over Direct Method

In the context of machine learning, FALCON achieved convolution performance greater than that of the direct method implemented in the industry-leading mathematical library for Intel Xeon Phi processors. The performance advantage of approximately 10% was measured for a workload simulating VGG Net forward pass. Based on an earlier argument for hybrid approach combining direct and fast algorithms (see chapter 5, figure 5.2), the speedup for this ConvNet may be improved to 30%. In some layers of VGG Net, FALCON is faster than MKL by as much as 50%, so the application of Winograd's algorithm to convolution in other DNN architectures may yield even more significant speedups.

7.4 Importance of High-Bandwidth Memory

According to comparison testing presented in chapter 5, high-bandwidth memory is the key element of the Intel Xeon Phi processor architecture that makes fast convolution perform better than the direct method. This is not an obvious result because ML tasks are generally considered compute-bound. However, as long as upcoming models of Intel Xeon Phi products retain the MCDRAM, they can benefit from fast convolution. In particular, performance advantage of fast convolution may develop strongly in the upcoming Knights Mill architecture specifically tuned for deep

learning applications top500 (2016) . In addition, the upcoming co-processor form-factor of Intel Xeon Phi co-processors is a suitable platform for ConvNets with fast convolution. Indeed, the data structures used for the VGG Net benchmark are under 16 GiB in size. This is suitable for offloading calculations to co-processors, assuming that they are manufactured with at least the same amount of MCDRAM as their bootable counterparts.

7.5 Application to Machine Learning

To my knowledge, the FALCON library is the first open-source implementation of fast convolution for Intel Xeon Phi processors. It is published under a permissive MIT license* in hopes that the high-performance computing community can contribute to the improvement of the code and to its adoption in production machine learning libraries.

Modern machine learning frameworks are layered, exposing a DNN interface to the computer scientist, but delegating convolution to an intermediate layer, and relying on GEMM in the underlying BLAS library. Therefore, regardless of the complexity of the fast or hybrid convolution, as long as it is implemented in the intermediate layer, ML application developers are going to experience performance improvement all the while retaining their code and computing solutions.

*github.com/ColfaxResearch/FALCON

Bibliography

Asai, R. (2016). Mcdram as high-bandwidth memory (hbm) in knights landing processors: Developer's guide. *Colfax Research*. 27

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759. 9, 10

Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. (2013). Deep learning with cots hpc systems. In Dasgupta, S. and Mcallester, D., editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1337–1345. JMLR Workshop and Conference Proceedings. 1

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. (2012). Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, pages 1223–1231, USA. Curran Associates Inc. 2

Karpathy, A. (2016). Cs231n convolutional neural networks for visual recognition. http://cs231n.github.io/convolutional-networks/. 7

Lavin, A. (2015). Fast algorithms for convolutional neural networks. *CoRR*, abs/1509.09308. 23

Ragate, S., Vladimirov, A., and Zhang, B. (2016). Falcon library: Fast image convolution in neural networks on intel architecture. *Colfax Research*. 2

Simonyan, K. and Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv e-prints*. 3

top500 (2016). Intel Unveils Plans for Knights Mill, a Xeon Phi for Deep Learning. 41

Winograd, S. (1980). *Arithmetic Complexity of Computations*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics. 17

Appendix

A Internal functions of Winograd Spatial Convolution

A.1 Input Transformation

```
// INTERNAL FUNCTION : FORM MATRIX A from input data, also includes
    transformation
static void get_tiles(const float* restrict image, const int ldi, const
    int irows, const int sizeI, const int C, float* restrict otile, const
    int N, const int ntiles){

    int t, u;
    #pragma omp parallel for
    for(t = 0; t < N*C; t++){

        int i, j, x;
        float tmp[16] __attribute__((aligned(64)));
        float s[16] __attribute__((aligned(64)));

        const float* data = image+t*sizeI;
        int tile_count = t*ntiles;

        // work on one image plane at a time, irrespective of the order
        for(i = 0; i < irows-2; i += 2){
            #pragma unroll(4)
            for(j = 0; j < (irows-2); j += 2){
                tmp[0 :4] =data[(i+0)*ldi+j:4];
                tmp[4 :4] =data[(i+1)*ldi+j:4];
                tmp[8 :4] =data[(i+2)*ldi+j:4];
```

```
tmp[12:4] =data[(i+3)*ldi+j:4];

// The tranformation manually simplified
s[0 ] =(tmp[0] - tmp[8 ]) - (tmp[2 ]- tmp[10]);
s[1 ] =(tmp[1] - tmp[9 ]) + (tmp[2 ]- tmp[10]);
s[2 ] =(tmp[2] - tmp[10]) - (tmp[1 ]- tmp[9 ]);
s[3 ] =(tmp[1] - tmp[9 ]) - (tmp[3 ]- tmp[11]);
s[4 ] =(tmp[4] + tmp[8 ]) - (tmp[6 ]+ tmp[10]);
s[5 ] =(tmp[5] + tmp[9 ]) + (tmp[6 ]+ tmp[10]);
s[6 ] =(tmp[6] + tmp[10]) - (tmp[5 ]+ tmp[9 ]);
s[7 ] =(tmp[5] + tmp[9 ]) - (tmp[7 ]+ tmp[11]);
s[8 ] =(tmp[8] - tmp[4 ]) - (tmp[10]- tmp[6 ]);
s[9 ] =(tmp[9] - tmp[5 ]) + (tmp[10]- tmp[6 ]);
s[10] =(tmp[10]- tmp[6 ]) - (tmp[9 ]- tmp[5 ]);
s[11] =(tmp[9] - tmp[5 ]) - (tmp[11]- tmp[7 ]);
s[12] =(tmp[4] - tmp[12]) - (tmp[6 ]- tmp[14]);
s[13] =(tmp[5] - tmp[13]) + (tmp[6 ]- tmp[14]);
s[14] =(tmp[6] - tmp[14]) - (tmp[5 ]- tmp[13]);
s[15] =(tmp[5] - tmp[13]) - (tmp[7 ]- tmp[15]);

// manually unrolled scatter to get max performance
otile[tile_count+0*STRIDE ] = s[0 ];
otile[tile_count+1*STRIDE ] = s[1 ];
otile[tile_count+2*STRIDE ] = s[2 ];
otile[tile_count+3*STRIDE ] = s[3 ];
otile[tile_count+4*STRIDE ] = s[4 ];
otile[tile_count+5*STRIDE ] = s[5 ];
otile[tile_count+6*STRIDE ] = s[6 ];
otile[tile_count+7*STRIDE ] = s[7 ];
otile[tile_count+8*STRIDE ] = s[8 ];
```

```
                otile[tile_count+9*STRIDE ] = s[9 ];
                otile[tile_count+10*STRIDE] = s[10];
                otile[tile_count+11*STRIDE] = s[11];
                otile[tile_count+12*STRIDE] = s[12];
                otile[tile_count+13*STRIDE] = s[13];
                otile[tile_count+14*STRIDE] = s[14];
                otile[tile_count+15*STRIDE] = s[15];



                tile_count++;
            }
        }
    }

}
```

A.2 Filter Transformation

```
// INTERNAL FUNCTION: FORM MATRIX B, also includes filter transform
static void filter_transform(const float* restrict filter, const int C,
    const int K, float* restrict out){

    int m, n, x;
    const float *F;

    #pragma omp parallel for collapse(2) private(m, n, x, F)
    for(m = 0; m < K; m++){
        for(n = 0; n < C; n++){
            float c1[16] __attribute__((aligned(64)));
```

```
F = filter+n*3*3 + m*3*3*C;

// work on in 3x3 plane at a time
// The tranformation manually simplified
c1[0] = F[0];
c1[1] = (F[0]+F[2]+F[1])*0.5f;
c1[2] = (F[0]+F[2]-F[1])*0.5f;
c1[3] = F[2];
c1[4] = (F[0]+F[6]+F[3])*0.5f;
c1[5] =
    ((F[0]+F[6]+F[3])+(F[2]+F[8]+F[5])+(F[1]+F[7]+F[4]))*0.25f;
c1[6] =
    ((F[0]+F[6]+F[3])+(F[2]+F[8]+F[5])-(F[1]+F[7]+F[4]))*0.25f;
c1[7] = (F[2]+F[8]+F[5])*0.5f;
c1[8] = (F[0]+F[6]-F[3])*0.5f;
c1[9] =
    ((F[0]+F[6]-F[3])+(F[2]+F[8]-F[5])+(F[1]+F[7]-F[4]))*0.25f;
c1[10] =
    ((F[0]+F[6]-F[3])+(F[2]+F[8]-F[5])-(F[1]+F[7]-F[4]))*0.25f;
c1[11] = (F[2]+F[8]-F[5])*0.5f;
c1[12] = F[6];
c1[13] = (F[6]+F[8]+F[7])*0.5f;
c1[14] = (F[6]+F[8]-F[7])*0.5f;
c1[15] = F[8];

// scatter
#pragma unroll(16)
for(x = 0; x < 16; x++){
    out[x*FSTRIDE+m*C+n] = c1[x];
}
```

```
        }
    }
}
```

A.3 GEMM

```
// INTERNAL FUNCTION
// GEMM specific to Ist layer of VGG with (M, N, K) = (12544, 64, 3)
// MKL performs bad
static void gemm_ker(int m, int n, int k, const float* a, const int lda,
    const float* b, const int ldb, float* c, const int ldc){

    const int BLK = 16;
    int x, xx, y, z, i;

    for(z = 0; z < n; z++){
        for(x = 0; x < m; x += BLK){
            float p[BLK] __attribute__((aligned(64)));
            p[0:BLK] = 0.0f;
            #pragma unroll(3)
            for(y = 0; y < 3; y++){
                #pragma vector aligned
                for(i = 0; i < BLK; i++){
                    p[i] += a[x+i+y*lda]*b[y+z*ldb];
                }
            }
            c[x+z*ldc:BLK] = p[0:BLK];
        }
    }
```

```
}


// INTERNAL FUNCTION
// C = A*B with beta = 0.0f and alpha = 1.0f
// Number of gemm calls is 16*BATCH
static void batched_gemm(const float* restrict image, const int irows,
    const int icols, const float* restrict filter, const int frows, const
    int fcols, float* restrict out, const int batch){

    int t, i;
    const char trans ='n';
    const float alpha = 1.0;
    const float beta = 0.0;
    const int ldi = irows;
    const int ldf = frows;
    const int ldo = irows;

    #pragma omp parallel for collapse(2) private(t, i)
    for(i = 0; i < 16; i++){
        for(t = 0; t < batch; t++){
            const float* im = image+i*STRIDE+t*irows*icols;
            const float* fi = filter+i*FSTRIDE;
            float* ot = out+i*STRIDE+t*irows*fcols;
            if(icols == 3) gemm_ker(irows, fcols, icols, im, ldi, fi, ldf,
                ot, ldo);
            else sgemm(&trans, &trans, &irows, &fcols, &icols, &alpha, im,
                &ldi, fi, &ldf, &beta, ot, &ldo);
        }
    }
```

```
}
```

A.4 Output Transformation

```
//OUTPUT TRANSFORM
static void out_transform(const float* restrict d, const int K, const int
    ntiles, float* restrict out, const int ldo, const int oH, const int
    oW, const int N){

    int t;
    int sizeO = oH*oW;

    #pragma omp parallel for
    for(t = 0; t < N*K; t++){

        float c1[16] __attribute__((aligned(64)));
        float temp[8] __attribute__((aligned(64)));
        float c2[4] __attribute__((aligned(64)));

        float* data = out +t*sizeO;
        int tile_offset = t*ntiles;

        int i, j;
        // work on one output plane at a time, irrespective of the order
        for(i = 0; i < oH; i += 2){
            for(j = 0; j < oW; j += 2){

                // gather the 16 elements form C to form a tile
                c1[0 ] = d[tile_offset+0 *STRIDE];
```

```
c1[1 ] = d[tile_offset+1 *STRIDE];
c1[2 ] = d[tile_offset+2 *STRIDE];
c1[3 ] = d[tile_offset+3 *STRIDE];
c1[4 ] = d[tile_offset+4 *STRIDE];
c1[5 ] = d[tile_offset+5 *STRIDE];
c1[6 ] = d[tile_offset+6 *STRIDE];
c1[7 ] = d[tile_offset+7 *STRIDE];
c1[8 ] = d[tile_offset+8 *STRIDE];
c1[9 ] = d[tile_offset+9 *STRIDE];
c1[10] = d[tile_offset+10*STRIDE];
c1[11] = d[tile_offset+11*STRIDE];
c1[12] = d[tile_offset+12*STRIDE];
c1[13] = d[tile_offset+13*STRIDE];
c1[14] = d[tile_offset+14*STRIDE];
c1[15] = d[tile_offset+15*STRIDE];

// The tranformation manually simplified
temp[0] = c1[0]+c1[1]+ c1[2];
temp[1] = c1[1]-c1[2]- c1[3];
temp[2] = c1[4]+c1[5]+ c1[6];
temp[3] = c1[5]-c1[6]- c1[7];
temp[4] = c1[8]+c1[9]+ c1[10];
temp[5] = c1[9]-c1[10]- c1[11];
temp[6] = c1[12]+c1[13]+ c1[14];
temp[7] = c1[13]-c1[14]- c1[15];

c2[0] = temp[0]+temp[2]+temp[4];
c2[1] = temp[1]+temp[3]+temp[5];
c2[2] = temp[2]-temp[4]-temp[6];
c2[3] = temp[3]-temp[5]-temp[7];
```

```
                data[i*ldo+j] =c2[0];
                data[i*ldo+j+1] =c2[1];
                data[(i+1)*ldo+j] = c2[2];
                data[(i+1)*ldo+j+1] = c2[3];
                tile_offset++;
            }
        }
    }
}
```

Vita

Sangamesh was born and raised in Bangalore, a city in southern India. He graduated from Visvesvaraya Technological University and earned his Bachelor's degree in Electronics & Communications Engineering. He worked as a design engineer implementing Digital Signal Processing algorithms on FPGA at a private organization in Bangalore for four years before coming to USA for higher studies. In 2014, he was enrolled in the Master of Science program with major in Computer Engineering at EECS department in the University of Tennessee, Knoxville. During his masters degree he was as a graduate research assistant at the Innovative Computing Laboratory working on performance analysis research in the PAPI group. In his spare time, he enjoys playing racquetball, cricket, reading about AI and solving simple sequence problems using recurrent neural network.