RICE UNIVERSITY

Optimizing Convolutions in State-of-the-art Convolutional Neural Networks on Intel Xeon Phi

by

Ankush Mandal

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Houston, Texas

July, 2017

ABSTRACT

Optimizing Convolutions in State-of-the-art Convolutional Neural Networks on Intel Xeon Phi

by

Ankush Mandal

Convolution layers are the core of Convolutional Neural Networks (CNNs), a class of Deep Neural Networks which provide state-of-the-art results in pattern recognition tasks in various domains, such as image recognition, speech recognition, and natural language processing. However, CNNs are very time consuming to train due to the computationally intensive nature of convolution operation. As a result, we have a prolific number of optimized implementations of convolution using matrix-matrix multiplication formulation, FFT formulation, Winograd algorithm, and direct convolution. Interestingly, most of these implementations target GPU architecture, which is mainly due to - 1) higher theoretical peak FLOPs of GPU and 2) popularity of CUDA deep learning library. However, not much investigation has been done on the performance of convolution on high-end CPUs such as Intel second generation Xeon Phi, codenamed Knights Landing (KNL) with theoretical peak performance of 6 TFLOPS. So, here we shed some light on what we can achieve regarding performance for convolution on these high-end CPUs. In this work, we optimize directed convolution for Intel Xeon Phi systems with AVX-512 support. Our strategy involves dynamic compilation approach along with standard compiler optimizations and software prefetching. We show that our JIT-based approach for direct convolution achieves close to peak performance on KNL for many cases. We also analyze

the performance of convolution layers of several state-of-the-art CNNs, pointing out what helps in performance gain and what breaks our approach.

Acknowledgments

First and foremost, I would like to convey my deepest gratitude to my advisor Dr. Vivek Sarkar for his mentorship, encouragement, and support throughout my graduate study at Rice University. His guidance and enthusiasm in research were invaluable to my development as a graduate student. He has been my source of inspiration whenever I faced difficulties. I owe him so much for giving me the opportunity to join Habanero Extreme Scale Software Research Group.

I would like to express the greatest degree of appreciation to my committee member Dr. Rajkishore Barik for giving me the opportunity to work with him at Intel. His expertise and insights were indispensable to the success of this project. I am incredibly grateful for his unwavering support during my time at Intel. I would also thank the team at Intel for all the help in this work.

I would like to thank my committee members Dr. Anshumali Shrivastava and Dr. Keith Cooper for their time and valuable feedbacks on this thesis.

I am grateful to all the members of the Habanero Extreme Scale Software Research group for their help and feedback in my research work. My study at Rice would not have been as entertaining and educational without them.

I have been very fortunate to have great friends in my graduate life. I am very thankful for all their support and encouragement. Special thanks to Ananya, Arijit, Arkabandhu, Arunim, Debarshi, Hamim, Himadri, Nilanjan, Pranabendu, Prasanth, Puspita, Rabimba, Rohan, Ronnie, Sagnik, Saikat, Sandip, Shams, Shantanu, Shuvadeep, Siam, Sourav, Sriraj, Trijit.

Last but not the least, I am indebted to my parents Kalpana Mandal and Durgapada Mandal, and my sister Ankita Mandal for their unconditional love. They continuously supported and encouraged me to pursue my dreams. Without them standing by my side, I could not have come this far.

Contents

Illustrations

Tables

Chapter 1

Introduction

Concepts from the field of Machine Learning drive many aspects of the modern society, from social networks to recommendations on e-commerce, and becoming more and more commercially valuable as we see an increasing number of consumer products powered by them such as cameras, self-driving car. In particular, Deep Learning has become one of the most critical technologies, demonstrating equal or even better than human-level performance for tasks in domains of object recognition, board games, and speech recognition. This became possible because due to two reasons - 1) its underlying model, Deep Neural Network (DNN) can learn features automatically from large datasets and represent complex functions using multiple hidden layers, 2) recent advances in processor technologies made it possible to satisfy the huge computational requirement associated with Deep Learning. Naturally, it is now an active topic of research in the community, leading many academic groups and companies release open source frameworks aiming to improve the productivity of data scientists by abstracting away implementation details. TensorFlow [3], Caffe [4] are few popular ones among such frameworks.

Although Different DNNs aim at different problems, one of the most important application today is image recognition [5] and currently, Convolution Neural Network (CNN) is the state-of-the-art DNN for this. The core of a CNN consists of multiple layers performing a large number of small convolutions. As an abundant amount of data parallelism is available in the computation of convolution through many images or mini-batch size and feature maps or channels, massively parallel architectures such as GPUs, in particular, has been used for training and inference on CNNs. As a consequence, all existing frameworks [3, 6, 7, 8, 4, 9] have GPU backend that implements convolution layers as libraries using cuDNN. However, high-end CPUs such as second-generation Intel Xeon Phi systems [10] with 70+ cores and AVX-512 support, has not been explored much in the domain of CNN. So in our perspective, it is noteworthy to shed some light on CPU performance and understand what it takes to achieve peak performance on CPUs for CNNs.

Although training and inference on CNNs are both time-consuming operations, requiring high-performance software implementations, training, in particular, takes an enormous amount of time and can span several weeks. For example, AlphaGo [11] took three weeks and 340 million training steps on 50 GPUs. So, we investigate into what makes training on CNN so time-consuming. We discover two factors here 1) training involves back propagation and weight update whereas inference does not, and it is much harder to have an optimized implementation for back propagation and weight update compared to forward propagation, a step common to both training and inference 2) usually training is done for large number of iterations compared to inference which is done in single pass. As the back propagation and weight update code are used over so many times during training, it makes sense to put some effort to optimize these steps.

Regarding convolution method, a general approach is to flatten the corresponding input data (image2column operation) and use standard matrix multiplications (GEMM) on the flattened data. This approach has a severe drawback. The flattening of input data is done to represent the image data as a 2-D matrix so that convolution can be formulated as GEMM. Hence, flattening step copies the input data from image data format (i.e. [Number of Images][Channel][Height][Width]) to 2-D matrix format, which is very low on arithmetic operations and makes the step purely memory bandwidth bound. Even though the GEMM computation is highly optimized, the flattening step acts as a bottleneck and creates a huge performance penalty. In recent years, a new and straightforward approach for implementation of convolution has started to materialize, called direct convolution. In this method, convolution is directly applied to input data.

In this work, we try to optimize back propagation and weight update on second-generation Intel Xeon Phi systems. We leverage direct convolution approach to avoid expensive memory operations and implement the convolutions associated with back propagation and weight update as direct convolution. [12] has shown that statically tuned BLAS calls incur overheads for small GEMMs and therefore do not achieve peak performance. They propose to use runtime code specialization via a JIT for small GEMMs. Since the tensor dimensions vary widely for convolutions on CNNs, we also propose runtime code specialization through a similar JIT strategy. We lay our data in a data format that helps the computation and apply standard compiler optimizations such as register blocking, cache blocking. When we mention ”data format”, we mean a particular sequence of the tensor dimensions. We also perform some optimizations specific to Xeon Phi, such as software prefetching. We demonstrate that, for training steps on CNNs, peak performance can be achieved on CPUs targeting High-Performance Computing (HPC) using this approach.

Chapter 2

Background

In this section, we describe the computation, i.e. convolution, and the architecture on which we are trying to optimize the computation, i.e. second-generation Intel Xeon Phi systems.

2.1 Convolution

In a convolution layer, each output pixel is generated from the weighted sum of a spatially connected neighborhood of input. So, convolution operation adds each element of the input image with elements from a defined neighborhood after multiplying all the elements with specific weights from filter data as shown in Figure 2.1.

In case of CNNs, we usually perform convolution over a batch of images. This is termed as batched convolution [13]. Mathematically, a batched convolution deals with three four-dimensional tensors: $I \in \boldsymbol{R}^{NCHW}$ as input image data, $O \in \boldsymbol{R}^{NKPQ}$ as output data, and $F \in \boldsymbol{R}^{KCRS}$ as filter data. The input data ranges over N images in a mini-batch, C input image feature maps, H rows or image height, W columns or image width. The filter data ranges over K output feature maps, C input feature maps, R rows or filter height, and S columns or filter width. The height and width of the output tensor is function of input image and filter height and width along with padding and stride, i.e. $P = f(H, R, stride\_h, pad\_h)$, and $Q = f(W, S, stride\_w, pad\_w)$. Here $stride\_h$ and $stride\_w$ are stride of access along respectively input image height and input image width. pad_h and pad_w are respectively height and width of the zero padding on each side of the input image. The

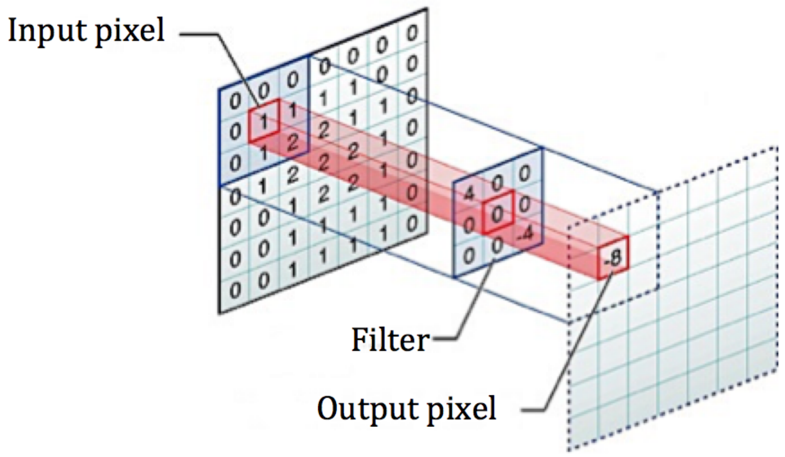

Figure 2.1 : Depiction of convolution (adapted from [1])

function f is defined as follows:

$$f(H, R, stride\_h, pad\_h) = \left\lceil \frac{H - R + 1 + 2 * pad\_h}{stride\_h} \right\rceil \tag{2.1}$$

Now, the convolution for back propagation can be defined as follows:

For $\forall n \in [0, N)\,, \forall c \in [0, C)\,, \forall h \in [0, H)\,, and\ \forall w \in [0, W)\,,$

$$\begin{aligned} I[n, c, h, w] = \sum_{k=0}^{K} \sum_{r=0}^{R} \sum_{s=0}^{S} & F[k, c, r, s].O[n, k, g(h, r, stride\_h, pad\_h), \\ & g(w, s, stride\_w, pad\_w)] \end{aligned} \tag{2.2}$$

Similarly, the convolution for weight update can be defined as follows:

For $\forall k \in [0, K)\,, \forall c \in [0, C)\,, \forall r \in [0, R)\,, and\ \forall s \in [0, S)\,,$

$$F[k, c, r, s] = \sum_{n=0}^{N} \sum_{p=0}^{P} \sum_{q=0}^{Q} I[n, c, p * stride\_h + r, q * stride\_w + s].O[n, k, p, q] \tag{2.3}$$

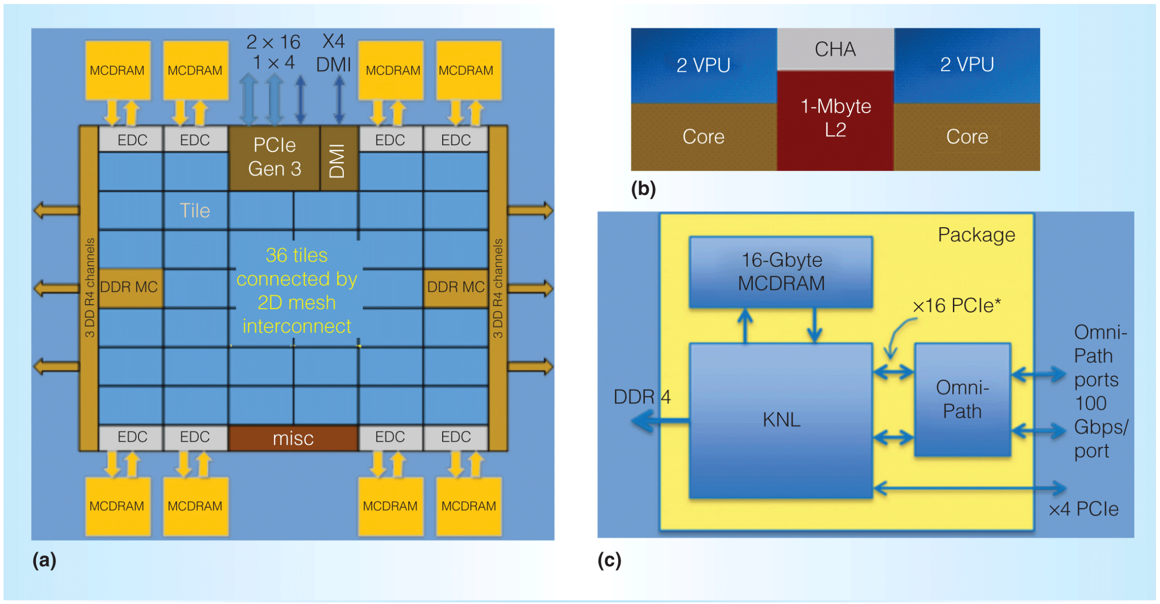

Figure 2.2 : KNL Architecture (adapted from [2])

2.2 Intel Xeon Phi systems

As discussed in chapter 1, we are interested in investigating what peak performance can be achieved on CPUs for convolutions in back propagation and weight update. As we are dealing with performance critical application, we are interested in high-end CPUs targeting HPC. Intel Xeon Phi processors are one family of such CPUs. The second generation Intel Xeon Phi processor code-named Knights Landing (KNL) is one of the strongest general purpose CPUs available today. So, we choose this processor as target architecture for our implementation and evaluate our implementation concerning performance over this processor.

A sample figure representing the architectural overview of KNL chip is given in Figure 2.2. The KNL chip features up to 72 out-of-order Silvermont Atom cores with each being 4-way hyper-threaded to hide memory and multi-cycle instruction latency. The cores are tiled in pairs and connected via 2-D mesh interconnect. In general, it is advisable to set one core aside for OS housekeeping. Micro-architecture

wise, one key feature of this processor is that each core is embedded with two 512-bit vector processing units (VPU) for increased SIMD level parallelism, i.e. each core can start execution of two 16-wide single precision SIMD instructions in the same clock cycle. Besides this, each core has 32 KB L1 data cache (3 cycles latency), 32 KB L1 instruction cache (3 cycles latency). Each tile has 1MB globally coherent L2 cache (17 cycle latency) shared between two cores. Apart from standard DDR4 memory up to 384 GB with a bandwidth of 90 GB/s, an additional 16 GB of high bandwidth stacked MCDRAM can be attached to the system that provides 500GB/s memory bandwidth. The latency for accessing data from memory can be as high as 160 ns.
Another important feature of KNL is that it supports explicit cache operation hint instructions. These include instructions to prefetch data into L1 or L2 caches (via prefetcht0 and prefetcht2 instructions respectively) and instruction modes to reduce the priority of a cache line. These instructions work together with streaming hardware L2 cache prefetcher. To achieve peak performance, one needs to carefully architect the placement and amount of L1/L2 prefetch instructions within the application.
It is important to mention that the machine is only two-issue wide but features two VPUs per core, which results in an upper limit of achievable peak performance as 80% of the theoretical peak performance. The reason behind this is the fact that, to achieve theoretical peak FLOPs, two issues have to be always filled with vector instructions of floating point operations. This is not possible in reality because there are overhead of loop management, pointer address calculations, and software prefetching to support corresponding memory movement operations. All of these instructions fill at least one of two pipeline slots and block a useful FMA instruction. Therefore, to achieve optimal performance on this system, these instructions should be limited to a bare minimum, which requires high-quality code concerning optimizations and can be quite a bit of challenge.

2.3 LIBXSMM JITer

We have previously mentioned in chapter 1 that we adopt a runtime code specialization strategy by using JIT compilation. In our work, we use LIBXSMM SMALLGEMM JITer [12] for generating optimized code for direct convolution at runtime. Originally, the JITer is designed for small matrix multiplications. It has an application specific GEMM code generation backend. Using this assembly generator for JITing, it showed significant performance improvement on x86 architectures. In order to generate code for direct convolution using this JITer, we modified the assembly generator backend. The reasons behind choosing this JITer are:

1) It is very light weight; it adds very little overhead during JITing. By compromising on generality, this JITer generates problem specific codes at runtime very efficiently. As we are targeting a particular problem, i.e. direct convolution, it makes sense to use a fast and low profile JITer. LIBXSMM SMALLGEMM JITer fits this criterion exactly.

2) It is designed for small matrix multiplication which follows fused multiply add pattern. As our direct convolution problem also follows fused multiply add pattern, this JITer is a good fit for our problem.

2) It targets AVX-512 systems, specifically Intel Xeon Phi systems. As our target is KNL, extending the AVX-512 code generation for direct convolution on KNL is easier.

We discuss in detail in section 3.4.4 about how we use this JITer and which goals we try to achieve during code generation.

Chapter 3

Optimizations for Direct Convolution on KNL

The input parameters for convolution including N, C, H, W, R, S, u or *stride_h*, and v or *stride_w* (section 2.1) vary significantly across layers of a CNN and also across different CNNs. This makes it hard to achieve peak IA performance by static compilation because the optimization factors and strategy depend on these parameters and these parameters are only known during execution time. For example, the loops to tile and their tiling factor, the loops to unroll and their unroll factor cannot be determined statically without knowing the values of these parameters. In order to address this problem, we propose a runtime code specialization approach to produce optimized code at runtime depending on the values of these parameters. This makes the core of our ninja code for direct convolution on IA.

In this section, we first discuss the nave direct convolution in back propagation and weight update. Then we discuss the data layouts for input, output, and weight tensors we considered and how they impact code optimization. We also give details on how we can efficiently implement direct convolution for back propagation and weight update on x86 architectures. For these ninja implementations, we discuss in particular about compiler optimizations such as register blocking, tiling, and unrolling to optimize the corresponding loop nests, runtime code specialization, code quality enhancements, and software prefetching.

```
for(img = 0; img < N; ++img) { //independent
  for(ifm = 0; ifm < C; ++ifm) { //independent
    for(ofm = 0; ofm < K; ++ofm) { //reduction
      for(oj = 0; oj < P; ++oj) { //carry dependency
        ij = oj * u;
        for(oi = 0; oi < Q; ++oi) { //carry dependency
          ii = oi * v;
          for(kj = 0; kj < R; ++kj) { //carry dependency
            for(ki = 0; ki < S; ++ki) { //carry dependency
              grad_input(img, ifm, ij + kj, ii + ki) +=
                grad_output(img, ofm, oj, oi) *
                weight(ifm, ofm, kj, ki);
 } } } } } } }
```

Figure 3.1 : Pseudo code of naive direct convolution for back propagation

3.1 Naive Direct Convolution for Back Propagation

A pseudo code of naive direct convolution for back propagation is given in Figure 3.1[*]. As we can see, it has seven deep loop nests. In the innermost loop, we read data from gradient output tensor and weight tensor. We multiply them and accumulate on gradient input tensor.

We annotate the independent loops as "independent". These loops are easily parallelizable. We also annotate loops carrying some dependency as "carry dependency" and loops across which reduction takes place as "reduction".

3.2 Naive Direct Convolution for Weight Update

A pseudo code of naive direct convolution for weight update is given in Figure 3.12. It also has seven deep loop nests like back propagation. But we do slight reordering of the loops to present the computation in a more clean way. In the innermost loop, we read data from gradient output tensor and input tensor. We multiply them and

[*]array accesses appear within "( )" instead of "[ ]" due to use of macros. A(i, j, k, l) = A [i*bound of(j)*bound of(k)*bound of(l) + j*bound of(k)*bound of(l) + k*bound of(l) + l]

```
for(ofm = 0; ofm < K; ++ofm) { //independent
  for(ifm = 0; ifm < C; ++ifm) { //independent
    for(img = 0; img < N; ++img) { //reduction
      for(oj = 0; oj < P; ++oj) {//reduction
        ij = oj * u;
        for(oi = 0; oi < Q; ++oi) {//reduction
          ii = oi * v;
          for(kj = 0; kj < R; ++kj) { //independent
            for(ki = 0; ki < S; ++ki) { //independent
              grad_weight(ofm, ifm, kj, ki) +=
                input(img, ifm, ij + kj, ii + ki) *
                grad_output(img, ofm, oj, oi);

 } } } } } } }
```

Figure 3.2 : Pseudo code of naive direct convolution for weight update

accumulate on gradient weight tensor. Furthermore, we also annotate the loops in the same way as we do in case of pseudo code for back propagation in Figure 3.1.

3.3 Data Layout

In this section, we consider different layouts of the tensor dimensions. For the reference of what they mean, we give a list of dimensions and their meaning in Table 3.1 and also a pictorial representation of input tensor in Figure 3.3.

The layout of input, output, and weight tensors play a significant role in achieving high throughput for convolution. In general, if the data access pattern matches the data layout, achieving spatial locality becomes much easier, and this is a crucial factor to consider for x86 architectures due to multiple levels of cache and TLB. Existing GPU frameworks such as cuDNN [13] use H and W dimension for vectorization, partly due to efficient scatter/gather support in the GPU architecture. However, from our experience, x86 architectures do not yet support efficient scatter/gather operations. Furthermore, the input feature map, C, and the output feature map, K, are

Table 3.1 : Tensor Dimensions

Dimensions	Description
N	Number of images
C	Input channel/feature map
K	Output channel/feature map
H	Input image height
W	Input image width
P	Output image height
Q	Output image width
R	Filter/Weight height
S	Filter/Weight width

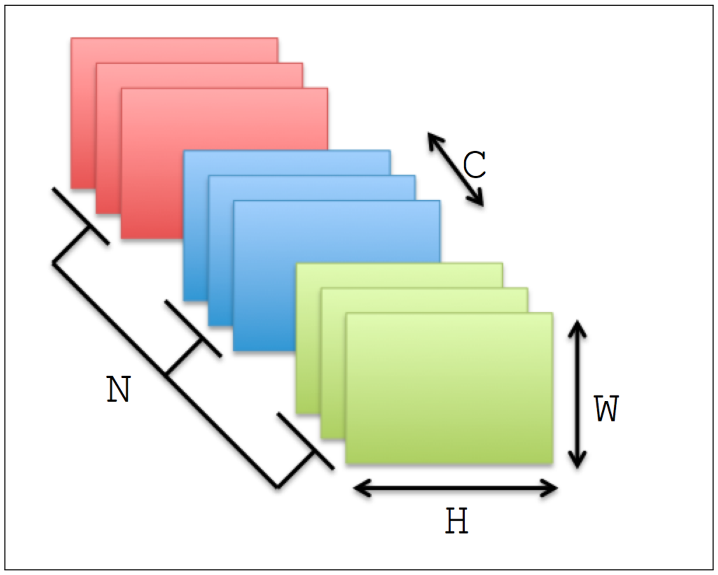

Figure 3.3 : Pictorial representation of input tensor

typically multiples of vector length on IA. So, for vectorization, we block these dimensions by vector length and bring in the blocking factor to the innermost dimension. Now, considering parallelization on back propagation, N and blocked C dimensions are good candidates because the loops corresponding to them are independent as we see in Figure 3.1. Furthermore, they usually have high value, which results in high level of parallelism in the code. So, it makes sense to make N and blocked C the outermost dimensions and parallelize the loops corresponding to these dimensions. Thus our data layout for input tensor becomes to $I \in NC_{B_I}HWB_I$ format where B_I denotes the blocking factor of input feature map and is equal to vector length. Similarly, the data layout for output tensor becomes to $O \in NK_{B_O}PQB_O$ format where B_O is the blocking factor of output feature map and also equals to vector length. We use same data layouts for input gradient and output gradient tensors.

Now for the weight tensor, which includes dimensions equal to input feature map and also output feature map, we have to decide the blocking factor from which dimension will be the innermost dimension. We use the blocking factor from output feature map for vectorization in both forward pass and weight update, and we only use the blocking factor of input feature map for vectorization in back propagation. So, based on usage frequency, we choose the blocking factor from output feature map as the innermost dimension for weight tensor. Considering parallelism for weight update, blocked C and blocked K are good candidates for parallelism as the loops corresponding to them are independent (Figure 3.12). So, we put them as the outermost dimensions. Thus the data layout for weight tensor becomes to $W \in K_{B_O}C_{B_I}RSB_IB_O$ format. The weight gradient tensor has similar data layout as weight tensor. We call these data formats collectively as Custom data format and all our optimizations refer to these data formats.

Besides using our Custom data format, we also experimented with the data format of a state-of-the-art DNN framework that is very popular in the field, namely, Tensorflow [3]. Tensorflow uses *NHWC* format for input and output tensors and *RSCK* format for weight tensor. By comparing performance across different data formats, we gain more insight into the impact of data layout on the performance of direct convolution on KNL.

3.4 Optimized Direct Convolution for Back Propagation on KNL

3.4.1 Vectorization

As we talked in section 3.3, our strategy for back propagation is to vectorize along the blocking factor of input feature map. So, we block the *ifm* loop in Figure 3.1 by a factor of vector length on IA and bring that blocking factor loop to the innermost position, i.e. after *ki* loop. Then we vectorize the loop. From the direct convolution loop nests for back propagation shown in Figure 3.1, we can easily see that the computation in the innermost loop follows multiply and accumulate computational pattern. For this computational pattern, we have several fused multiply and add (**fmadd**) vector instructions [14]. For example, **fmadd** operation on 32-bit floating-point data comes with these variants: **vfmadd132ps**, **vfmadd213ps**, **vfmadd231ps**, and **vfmadd233ps**. The first three variants can be represented as follows:

```
vfmadd132ps::zmm1,zmm2,zmm3 :  zmm1 = zmm1xzmm3+zmm2
vfmadd213ps::zmm1,zmm2,zmm3 :  zmm1 = zmm2xzmm1+zmm3
vfmadd231ps::zmm1,zmm2,zmm3 :  zmm1 = zmm2xzmm3+zmm1
```

Where **zmm** represents vector registers, "**::**" represents argument list to a instruction, and "**:**" represents translation of a instruction to mathematical operation. Memory loads, modifiers such as conversion or broadcast are only applied to **zmm3**. The fourth

`fmadd` variant, i.e. **`vfmadd233ps`** is a special **`fmadd`** instruction that acts as scale and bias transformation in one instruction. As we have separate tensors for input, output, and weight, this instruction is not applicable in our case.

Considering the **`fmadd`** vector instructions, we think **`vfmadd231ps`** is the most suitable instruction in our case. We implement our multiply and accumulate computation with the following strategy:

- As the data layout of input tensor has B_I as innermost dimension and it is of vector length, we achieve vectorizing accesses to input. We use **`zmm1`** for input because we want to accumulate to input.

- Now, the weight tensor has B_O as the innermost dimension and then B_I. So, in order to vectorize along B_I dimension, we transpose weight tensor so that B_I becomes the innermost dimension. This adds an overhead but if we have to maintain the same data layout for forward pass, back propagation, and weight update, we have pay some penalty in one of the stages because all of the stages do not use same dimension for vectorization. We use **`zmm2`** for weight.

- The output is then broadcast to a vector register in order to perform vector multiplication. As we need broadcast operation for output and we can use only **`zmm3`** for broadcast, we use **`zmm3`** for output.

So, we perform the multiply and add operation inside our innermost loop in Figure 3.1 with combination of **`broadcast`** and **`fmadd`** in a single vector instruction, namely, **`vfmadd231ps`**. The resulting code after vectorization is shown in Figure 3.4.

3.4.2 Optimize for Temporal and Spatial Locality

Temporal locality and spatial locality are very important factors for achieving peak performance in x86 architectures because of multiple level cache hierarchy and TLB.

```
for(img = 0; img < N; ++img) {
 for(ifm1 = 0; ifm1 < C/VLEN; ++ifm1) { //blocked by VLEN
  for(ofm1 = 0; ofm1 < K/$B_O$; ++ofm1) {
   for(ofm2 = 0; ofm2 < $B_O$; ++ofm2) {
    for(oj = 0; oj < P; ++oj) {
     ij = oj * u;
     for(oi = 0; oi < Q; ++oi) {
      ii = oi * v;
      for(kj = 0; kj < R; ++kj) {
       for(ki = 0; ki < S; ++ki) {
        grad_input_vec = vload(gard_input[img][ifm1][ij+kj]
            [ii+ki][0]);
        weight_vec = vload(weight[ofm1][ifm1][kj][ki][ofm2][0]);
        vfmadd231ps::grad_input_vec,weight_vec,
              broadcast(grad_output[img][ofm1][oj][oi][ofm2]);
        vstore(gard_input[img][ifm1][ij+kj][ii+ki][0],
            grad_input_vec);
 } } } } } } } }
```

Figure 3.4 : Pseudo code for back propagation after vectorization

The code in Figure 3.4 that we get after vectorization can be optimized for temporal locality by blocking for floating-point vector registers (register blocking). If we consider the data layout of input tensor, it becomes apparent that W dimension is ideal candidate for register blocking since it is the innermost dimension after B_I which is used for vectorization. However, W dimension changes with `oi` and `ki` loops. If we block along `oi` loop, we will achieve temporal locality for weight tensor since it is invariant to `oi` loop. On the other hand, if we block along `ki` loop, we will gain temporal locality for output tensor. In case of CNNs, the filter width, S, usually has small value and the output feature width, Q, usually have large enough value. Hence, in order to gain enough temporal locality through register blocking we use `oi` loop. We unroll-and-jam `oi` loop and decide the unroll factor based on input parameter values and number of available vector registers. The resulting pseudo code after register blocking is shown in Figure 3.5. The constraint for the unroll factor here is the number of available floating-point vector registers (typically 32 in modern

```
for(img = 0; img < N; ++img) {
 for(ifm1 = 0; ifm1 < C/VLEN; ++ifm1) {
  for(ofm1 = 0; ofm1 < K/$B_O$; ++ofm1) {
   for(ofm2 = 0; ofm2 < $B_O$; ++ofm2) {
    for(oj = 0; oj < P; ++oj) {
     ij = oj * u;
     for(oi = 0; oi < Q/$B_Q$; ++oi) { //$B_Q$ is equal to
         unroll_factor_rb
      ii = oi * v;
      for(kj = 0; kj < R; ++kj) {
       for(ki = 0; ki < S; ++ki) {
        weight_vec = vload(weight[ofm1][ifm1][kj][ki][ofm2][0]);

        grad_input_vec = vload(gard_input[img][ifm1][ij+kj]
            [oi*$B_Q$*v+ki][0]);
        vfmadd231ps::grad_input_vec,weight_vec,
            broadcast(grad_output[img][ofm1][oj][oi*$B_Q$][ofm2]);
        vstore(gard_input[img][ifm1][ij+kj][oi*$B_Q$*v+ki][0],
            grad_input_vec);

        grad_input_vec = vload(gard_input[img][ifm1]
            [ij+kj][(oi*$B_Q$+1)*v+ki][0]);
        vfmadd231ps::grad_input_vec,weight_vec,
            broadcast(grad_output[img][ofm1][oj][oi*$B_Q$+1][ofm2]);
        vstore(gard_input[img][ifm1]
            [ij+kj][(oi*$B_Q$+1)*v+ki][0],grad_input_vec);

        ...//$B_Q$ times
} } } } } } } }
```

Figure 3.5 : Pseudo code for back propagation after 1-D register blocking

x86 architectures). However, we set aside at least 4 floating-point vector registers for other purposes (described later). Then the number of floating-point vector registers available for register blocking becomes 28. So, we set the unroll factor as follows:

`unroll_factor_rb = min(Q, divider of Q lower than 28)`

where rb denotes register blocking. Typically the value of Q is high enough that 1-D register blocking suffices. If the value of Q is so small that 1-D register blocking does not suffice, we go for 2-D register blocking which includes another level of register blocking along oj loop.

The code from Figure 3.5 can be further optimized to gain spatial locality and more temporal locality by blocking for L1 data cache (cache blocking). Since we are broadcasting one element of output tensor at each **`fmadd`** instruction and the innermost dimension of output tensor is B_O, we can gain spatial locality in output tensor access by bringing *`ofm2`* loop inside *`ki`* loop. Here, we make the assumption that `unroll_factor_rb` * B_O output element fits in the L1 cache. In our case, `unroll_factor_rb` $\leq$ `28` and B_O is *`VLEN`* which is typically a cache line size on IA for AVX-512. Since we are considering 28 cache lines and L1 cache can contain up to 512 cache lines, our assumption is quite reasonable. Besides bringing spatial locality, this L1 cache blocking along *`ofm2`* loop also brings temporal locality in input tensor access because accumulation occurs on the same input values along *`ofm2`* loop. The resulting code after L1 cache blocking is given in Figure 3.6.

3.4.3 Optimize loads and stores

We do vector load operation on weight and input tensor and vector store operation on input tensor. We do not perform vector load on output tensor, but we perform broadcast instead, i.e. one element of output tensor is broadcast to all the lanes of vector register containing values from weight tensor. The code from Figure 6 can be further optimized regarding loads and stores. First, we consider the vector loads of weight tensor. The **`fmadd`** instructions are dependent on the weight vector load that precedes them. Therefore a cache miss on weight load can cause havoc on performance because it will stall all the **`fmadd`** instructions following it. In order to eliminate such stalls, we schedule the weight vector loads in advance at the cost of a moderate increase in register pressure. We use 4 floating-point vector registers for loading weights in advance and setting up a 4-register ring-buffer of vector loads on weight tensor.

```
for(img = 0; img < N; ++img) {
 for(ifm1 = 0; ifm1 < C/VLEN; ++ifm1) {
  for(ofm1 = 0; ofm1 < K/$B_O$; ++ofm1) {
   for(oj = 0; oj < P; ++oj) {
    ij = oj * u;
    for(oi = 0; oi < Q/$B_Q$; ++oi) { //$B_Q$ is equal to
       unroll_factor_rb
     ii = oi * v;
     for(kj = 0; kj < R; ++kj) {
      for(ki = 0; ki < S; ++ki) {
       { weight_vec = vload(weight[ofm1][ifm1][kj][ki][0][0]);

         grad_input_vec = vload(gard_input[img][ifm1][ij+kj]
             [oi*$B_Q$*v+ki][0]);
         vfmadd231ps::grad_input_vec,weight_vec,
             broadcast(grad_output[img][ofm1][oj][oi*$B_Q$][0]);
         vstore(gard_input[img][ifm1][ij+kj][oi*$B_Q$*v+ki][0],
             grad_input_vec);

         grad_input_vec = vload(gard_input[img][ifm1]
             [ij+kj][(oi*$B_Q$+1)*v+ki][0]);
         vfmadd231ps::grad_input_vec,weight_vec,
             broadcast(grad_output[img][ofm1][oj][oi*$B_Q$+1][0]);
         vstore(gard_input[img][ifm1]
             [ij+kj][(oi*$B_Q$+1)*v+ki][0],grad_input_vec);

         ... //$B_Q$ times
        }

       { weight_vec = vload(weight[ofm1][ifm1][kj][ki][1][0]);

         grad_input_vec = vload(gard_input[img][ifm1][ij+kj]
             [oi*$B_Q$*v+ki][0]);
         vfmadd231ps::grad_input_vec,weight_vec,
             broadcast(grad_output[img][ofm1][oj][oi*$B_Q$][1]);
         vstore(gard_input[img][ifm1][ij+kj][oi*$B_Q$*v+ki][0],
             grad_input_vec);

         grad_input_vec = vload(gard_input[img][ifm1]
             [ij+kj][(oi*$B_Q$+1)*v+ki][0]);
         vfmadd231ps::grad_input_vec,weight_vec,
             broadcast(grad_output[img][ofm1][oj][oi*$B_Q$+1][1]);
         vstore(gard_input[img][ifm1]
             [ij+kj][(oi*$B_Q$+1)*v+ki][0],grad_input_vec);

         ... //$B_Q$ times
        }

        ... //$B_O$ times
} } } } } } } }
```

Figure 3.6 : Pseudo code for back propagation after L1 cache blocking

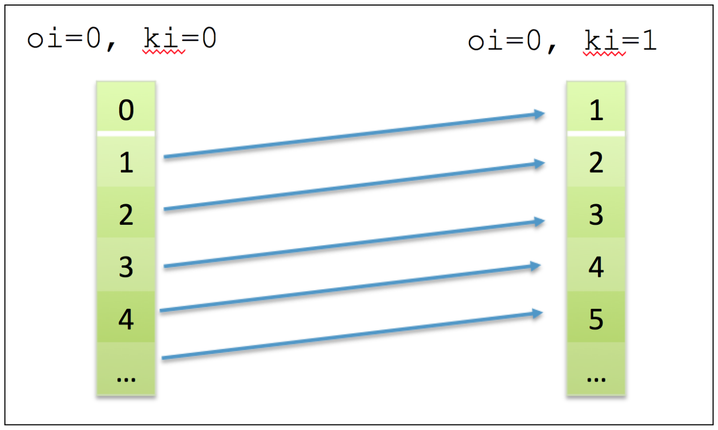

Figure 3.7 : Access pattern for input when stride is 1

The scenario for vector load and store on input tensor is little tricky. From Figure 3.6, we see that we need `unroll_factor_rb` number of vector length inputs in each iteration of `ki`. However, after carefully analyzing input access pattern along W dimension, we see that, as `ki` varies from one iteration to another iteration, there is a significant reuse in input access if the stride along W dimension is one. The scenario is depicted in Figure 3.7. In case of CNNs, the stride is one most of the times. Hence, if we take advantage of the reuse of input data, we can reduce the number of vector loads and stores significantly on input data. The strategy we use is as follows:

- In the first iteration of `ki`, we load all the `unroll_factor_rb` vector length inputs. Lets index them by 0 to `(unroll_factor_rb-1)`
- Then for the next iteration, we need vector length inputs from 1 to `unroll_factor_rb`. So, from next iteration of `ki`, we load only one vector length input that has the next index to the last index from the previous iteration.
- We shift and rotate the register indices so that **`fmadds`** are performed on corresponding right floating-point vector registers.
- We store the vector register that has the first index for the current iteration. If

it is the last iteration of `ki`, we store all the vector registers.

We unroll `ki` loop completely and employ the load/store strategy mentioned above. Moreover, it also reduces branching in the code. Although the value of R and S are usually small, typically we unroll only `ki` loop. This is because the code needs to fit in the L1 instruction cache. We discuss this in detail in section 3.4.5. After optimizing the vector loads and stores, the pseudo code becomes the one given in Figure 3.8.

3.4.4 Runtime Code Specialization and Parallelization

As we discussed at the beginning of chapter 3, we cannot achieve peak performance through static compilation in case of direct convolution on CNNs because of widely varying values of input parameters. Our optimization strategy and factors depend on these input parameter values, and these dynamic values are only known at execution time. So, our approach is to perform runtime code specialization, and we do so with the help of SMALLGEMM jus-in-time assembly generator [12].

The innermost code sequence starting from `kj` loop in Figure 3.8 is abstracted out to the SMALLGEMM JITer. The assembly code for this code sequence is generated before entering the loop nest as shown in Figure 3.9. The `bp_jit` function pointer takes the convolution parameters as a descriptor and generates an optimal sequence of x86 instructions. Since we generate the JITed code only at the setup phase and the cost of code generation by SMALLGEMM is very low, there is virtually no overhead of JITing in our case. Furthermore, we can generate more than one version of specialized JITed code and use the appropriate version in the corresponding scenario. However, there are two constraints when aiming for peak performance with this approach. We have to ensure that the code fits in L1 instruction cache while switching between multiple versions of the JITed code and the memory footprint of each JITed code fits in the L1 data cache. Fortunately, it is easy to do so in our case. As we keep the loop

```
for(img = 0; img < N; ++img) {
 for(ifm1 = 0; ifm1 < C/VLEN; ++ifm1) {
  for(ofm1 = 0; ofm1 < K/$B_O$; ++ofm1) {
    for(oj = 0; oj < P; ++oj) {
     ij = oj * u;
     for(oi = 0; oi < Q/$B_Q$; ++oi) {
      ii = oi * v;
      for(kj = 0; kj < R; ++kj) {
       for(ki = 0; ki < S; ++ki) { //unrolled, but represented
          //here as loop to give cleaner logical view of the code
        weight_vec_0 = vload(weight[ofm1][ifm1][kj][ki][0][0]);
        weight_vec_1 = vload(weight[ofm1][ifm1][kj][ki][1][0]);
        weight_vec_2 = vload(weight[ofm1][ifm1][kj][ki][2][0]);
        weight_vec_3 = vload(weight[ofm1][ifm1][kj][ki][3][0]);

        if (ki == 0) {
         vload $B_Q$ VLEN grad_inputs starting from
            grad_input[img][ifm1][ij+kj][oi*$B_Q$*v][0];
        } else {
         vload (grad_input[img][ifm1][ij+kj][(oi*$B_Q$+($B_Q$+ki-1))*v
            ][0]);
        }

        { //starting index for grad_input_vec_reg is determined by
          //rotating register indices
        vfmadd231ps::grad_input_vec_reg_start,weight_vec_0,
           broadcast(grad_output[img][ofm1][oj][oi*$B_Q$][0]);
        vfmadd231ps::grad_input_vec_reg_start+1,weight_vec_0,
           broadcast(grad_output[img][ofm1][oj][oi*$B_Q$+1][0]);
        ... }

        {
        weight_vec_4 = vload(weight[ofm1][ifm1][kj][ki][4][0]);

        vfmadd231ps::grad_input_vec_reg_start,weight_vec_1,
           broadcast(grad_output[img][ofm1][oj][oi*$B_Q$][1]);
        vfmadd231ps::grad_input_vec_reg_start+1,weight_vec_1,
           broadcast(grad_output[img][ofm1][oj][oi*$B_Q$+1][1]);
        ...  }
        ...
        if (ki == (S-1)) {
         vstore $B_Q$ grad_input_regs;
        } else {
         vstore grad_input_vec_reg_start;
        }
 } } } } } } }
```

Figure 3.8 : Pseudo code for back propagation after optimization of loads and stores

```
bp_desc = setup backward descriptor using N,C,H,W,K,R,S,u,v,pad_h,
    pad_w;
conv_bp = bp_jit(bp_desc, ...); //Call to JITer to generate JIT
    code at Runtime
#pragma omp for collapse(2)
for(img = 0; img < N; ++img) {
 for(ifm1 = 0; ifm1 < C/VLEN; ++ifm1) {
  for(ofm1 = 0; ofm1 < K/$B_O$; ++ofm1) {
   for(oj = 0; oj < P; ++oj) {
    ij = oj * u;
    for(oi = 0; oi < Q/$B_Q$; ++oi) {
     float *inp_ptr = &(grad_input[img][ifm1][ij][oi*$B_Q$*v][0]);
     const float *out_ptr = &(grad_output[img][ofm1][oj]
         [oi*$B_Q$][0]);
     const float *wt_ptr = &(weight[ofm1][ifm1][0][0][0][0]);
     conv_bp(input_ptr, wt_ptr, out_ptr);
} } } } }
```

Figure 3.9 : Pseudo code for back propagation after runtime code specialization and parallelization

nests for register and cache blocking outside of our JITer, we can control the code size and memory footprint fairly easily. The memory footprint of *bp_jit* in bytes can be computed as follows:

$$\begin{aligned} Mem(\texttt{bp\_jit}) = {} & B_Q * B_O * 4 \ (for\ output) + \\ & R * S * B_O * VLEN * 4 \ (for\ weight) + \\ & R * (B_O + S) * VLEN * 4 \ (for\ input) \end{aligned} \tag{3.1}$$

The parameters B_O, B_Q, unroll factor of `ki` loop, and unroll factor of `kj` loop are chosen such that $Mem(\texttt{bp\_jit})$ is less than L1 data cache size (typically 32 KB in Intel x86 Architectures). Moreover, we also ensure that the code size of *bp_jit* and all of its variants generated at runtime is less than L1 instruction cache size (typically 32 KB in IA). We discuss this in detail in section 3.4.5.

As we can see in Figure 3.9, we dedicate our JITer to generate highly optimized code only for the most critical innermost loop nests. We still leverage a regular compiler for parallelizing and optimizing outer loop nests. Having the outer loop nests in high-level language has several advantages. They can be easily parallelized using widely available threading libraries, such as OpenMP. In our case, the outermost `img` loop and blocked `ifm1` loop are independent and easily parallelizable. Furthermore, the mini-batch size, *N*, usually have high value. So, the mini-batch loop together with `ifm1` loop exhibits sufficient thread-level parallelism for x86 architectures. Hence, we use them to exploit thread-level parallelism via OpenMP constructs such as **`collapse`** and **`pragma omp parallel`**.

3.4.5 Optimize Code Size

In order to achieve peak performance on x86 architecture, we have to ensure that the code fits in L1 instruction cache. Otherwise, we have to pay a huge penalty for fetching it from lower level caches or main memory. The parameters that affect JITed code size is B_O, B_Q, unroll factor of `ki` loop, and unroll factor of `kj` loop. Now for determining code size, we take a nave approach. We use a simple heuristic where we have an empirical value of the code size when all these parameters have a value of 1. Now we multiply that code size with the values of these parameters and get a projection of the final JITed code size. Depending on the values of code size, we determine whether to unroll `ki` loop and or `kj` loop. On KNL, we typically unroll only `ki` loop.

Besides these parameters, there is also another issue that can increase the code size and negatively affect the performance. As Xeon Phi does not have a decoded instruction cache, AVX-512 code quality plays an important role in achieving good performance. A general rule that we have to follow in this case is that only up to 16 Bytes can be fetched per cycle from the instruction cache. This constraint becomes problematic in the case of our **`fmadd`** operations. When reading output in a broadcast fashion, we have strided access by a factor of B_O. This sometimes results in an offset larger than 1024 bytes between broadcasts. In such a case, the length of the **`fmadd`** instruction increases from 7 to 10 bytes and instructions with this increased length cannot be fetched on a sustainable basis from the instruction cache. So, we utilize x86 SIB (scale index base) addressing mode to reduce the instruction size to 8 bytes per **`fmadd`** operation. Since we have spare general purpose (gp) registers, we use them for expressing up to 28 streams. The strategy is as follows:

- In SIB mode an address is represented as $(Base\ Address + Scale * Index)$
- We load the size of B_O in a gp register. We also store 3*B_O, 5*B_O, and 7*B_O in three other gp registers. We use these registers as index registers.
- Due to scaling in SIB mode, {1,2,4,8} * any index register value can be added for free. Hence, we are now able to express 9 addresses with stride B_O starting from a *Base Address*.
- Further storing $BaseAddress + \{9, 18, 27\} * B_O$ in three different gp registers, we are now able to express up to 28 addresses with the stride of B_O.

3.4.6 Hiding fmadd Latency

As discussed in section 2.2, each KNL core has two VPUs. The latency of each VPU is 6 cycles. So, if our vector floating-point register blocking is less than 12, we issue a **`fmadd`** instruction to a VPU before it has finished executing the previous **`fmadd`** instruction on a vector register. This creates **`fmadd`** stalls and bubble in the pipeline. To avoid this kind of situation, we proposed in section 3.4.2 a 2-D register blocking, i.e. another level of register blocking along H dimension. However, we cannot make register blocking arbitrarily large. We have the upper limit set to 28 vector floating-pointer registers for this purpose. Sometimes the values of H and W may prevent us doing 2-D register blocking within this constraint. For example, the scenario where $H = 7$ and $W = 7$. In this kind of scenario, we use another strategy to hide **`fmadd`** latency. We use extra accumulator registers and store partial results alternatively in those accumulators. At the end of the convolution, we sum up the result to get the final value of a register. This strategy transforms dependent instructions to independent instructions as we have decoupled the destination registers.

3.4.7 Software Prefetch

Intel x86 ISA has explicit L1 and L2 cache prefetch instructions in the form of **`prefetcht0`** and **`prefetcht2`**, respectively. We use these instructions in our JITer to prefetch the required data from one JITed function call to another. We extended the JITer interface with additional arguments that hold the address for input, output, and weight tensor accesses for the next JITed function call. As we discussed in section 3.4.3, we schedule weight loads in advance. So, our prefetch strategy is to prefetch the weights to L2 cache, but prefetch input and output to L1 cache. The input prefetches are inserted along the input vector loads and that makes total $R*(B_O+S)$ input prefetches, which amounts to $R*(B_O+S)*64$ bytes of input data being brought to L1 data cache. This is because a single prefetch instruction brings one cache line size of data and cache line size in IA is typically 64 bytes. We insert the output prefetches along **`fmadd`** instructions until all the required B_Q cache lines are prefetched. This results in B_Q*64 bytes of output data being brought to L1 data cache. We also insert the weight prefetches along **`fmadd`** instructions, which results in $R*S*B_O*64$ bytes of weight data being prefethed to L2 cache.

We can see from Figure 3.9, the weight tensor is reused across `oi` and `oj` loop. Since weights are reused, there is no need to prefetch weight inside `oi` and `oj` loop.

```
bp_desc = setup backward descriptor using N,C,H,W,K,R,S,u,v,pad_h,
   pad_w;
conv_bp_pf_noweight = bp_jit(bp_desc, PREFETCH_NO_WEIGHT); //JITed
    code without weight prefecth
conv_bp_pf = bp_jit(bp_desc, PREFETCH_ALL); //JITed code with
   weight prefecth
#pragma omp for collapse(2)
for(img = 0; img < N; ++img) {
 for(ifm1 = 0; ifm1 < C/VLEN; ++ifm1) {
  for(ofm1 = 0; ofm1 < K/$B_O$; ++ofm1) {
   for(oj = 0; oj < P; ++oj) {
    ij = oj * u;
    for(oi = 0; oi < Q/$B_Q$; ++oi) {
     float *inp_ptr = &(grad_input[img][ifm1][ij][oi*$B_Q$*v][0]);
     const float *out_ptr = &(grad_output[img][ofm1][oj]
         [oi*$B_Q$][0]);
     const float *wt_ptr = &(weight[ofm1][ifm1][0][0][0][0]);
     if ( not last iteration of oj loop) {
      //do not prefetch weights
      conv_bp_pf_noweight(input_ptr, wt_ptr, out_ptr,
                   &(grad_input[img][ifm1][ij][(oi+1)*$B_Q$*v][0]),
                   NULL, //weights are not prefetched
                   &(grad_output[img][ofm1][oj][(oi+1)*$B_Q$][0]);

    } else { //last iteration of oj loop, prefetch weights
      conv_bp_pf(input_ptr, wt_ptr, out_ptr,
                &(grad_input[img][ifm1][0][0][0]),
                &(weight[ofm1+1][ifm1][0][0])
                &(grad_output[img][ofm1+1][0][0]);
    }
} } } } }
```

Figure 3.10 : Pseudo code for back propagation with JIT interface and software prefetching

We only need to prefetch weight at the end of oj loop. So, to prevent redundant weight prefetch, we generate two versions of JITed code, one without weight prefetch (*conv_bp_pf_noweight*) and another with weight prefetch (*conv_bp_pf*). The resulting code with software prefetching is shown in Figure 3.10.

3.4.8 Low Precision Operations

In general, the data associated with CNNs are represented in 32-bit single precision floating points. But, currently, there is a growing interest within the community to present the data associated with CNNs in low precision format and perform convolution on these low precision data to accelerate the computation. We also investigated on this topic of performing convolution efficiently for low precision data, on Intel x86 architecture. Intel Software Developers Manual March 2017 update [15] includes some instructions that are of interest considering our objective here.

First, let us consider the low precision formats and configurations that we can have 1) 16-bit data and 32-bit accumulator, 2) 8-bit data and 16-bit accumulator, 3) 8-bit data and 32-bit accumulator. The general optimization strategies that we talked earlier remain more or less same for convolution in these low precision formats, except the strategy to perform fused multiply-adds (FMA) inside the innermost loop. So, we present our revised strategy to perform FMA in these low precision formats here.

1) 16-bit data and 32-bit accumulator In general for low precision computation, we reserve one floating point vector register storing pairs of output data, and one other for storing the result of partial sum. As a result, we have 26 vector registers for register blocking instead of 28 vector registers.

- We convert input data from 16-bit to 32-bit before passing the input tensor pointer to the JITed function.
- During weight loads, we load 32 16-bit weight elements into a vector register.
- For performing FMA, we first broadcast a pair of 16-bit values from output tensor to one of our reserved vector register using **`VPBROADCASTD`** instruction.
- Then we perform 16-bit multiply and add with horizontal add using **`VPMADDWD`** instruction. It multiplies the individual signed words of the destination operand

with the corresponding signed words of the source operand, producing temporary signed doubleword results. The adjacent doubleword results are then summed (we refer to this as horizontal add) and stored in the destination operand, the other reserved vector register used for storing partial sums.

- We then add this result to vector register containing input tensor values. We use **VPADDD** instruction for this.
- After returning from of the JITed function call, we convert input data from 32-bit to 16-bit.

The whole process is depicted in Figure 3.11.

2) 8-bit data and 16-bit accumulator the strategy is exactly same as the one described for 16-bit data and 32-bit accumulator. The only exceptions are - We use **VPBROADCASTW** instruction for broadcasting pairs of 8-bit output values, **VPMADDUBSW** instruction for doing FMA with horizontal add, and **VPADDW** instruction for adding the intermediate result to input vector register.

3) 8-bit data and 32-bit accumulator the strategy, in this case, is similar to what we stated for 16-bit data and 32-bit accumulator scenario. However, we need one more vector register to perform horizontal add between intermediate 16-bit results. So, now we are left with 25 vector registers for register blocking.

- We convert input data from 8-bit to 32-bit before calling the JITed function.
- During weight loads, we load 64 8-bit weight tensor values into a vector register.
- We broadcast quadruple of 8-bit output tensor values to a vector register.
- Then we perform 8-bit FMA with horizontal add using **VPMADDUBSW** instruction which produces 16-bit intermediate results.

- We further perform horizontal add of these results. For this, we preload one vector register with 32 16-bit 1s. We use this vector register to perform FMA with horizontal add on 16-bit intermediate values which results in 32-bit horizontal add because one operand of the multiplications is 1. We use **`VPMADDWD`** instruction for this purpose.
- Then we add these 32-bit partial sums to input register using **`VPADDD`** instruction.
- After returning from JITed function, we convert back input data from 32-bit to 8-bit.

3.4.9 Packed fmadd Operations

Another interesting class of instructions from Intel x86 ISA [16] that is relevant to our case is packed single-precision floating-point **`fmadd`** instructions. We are specifically interested in **`V4FMADDPS`** instruction. This instruction computes four sequential packed fused multiply add instructions with a sequentially selected memory operand in each of the four steps. The instruction works as follows:

`V4FMADDPS :: zmm1, zmm2+3, m128` - it accesses four source registers starting with **`zmm2`**; sources are consecutive and start in a multiple-of-4 boundary. It multiplies packed single-precision floating-point values from these source registers by values from **`m128`** and accumulates the result in **`zmm1`**.

We use this instruction to execute efficiently four **`fmadd`**s together in a single instruction if the blocking factor B_O (Figure 3.10) of output feature map is divisible by 4, which is typically the case. We replace each sequence of 4 **`vfmadd231ps`** instructions along BO with one **`V4FMADDPS`**. One downside of this approach, we cannot set up a 4-register ring-buffer for weight loads as described in section 3.4.3 because now we need four source registers for weight data in each step.

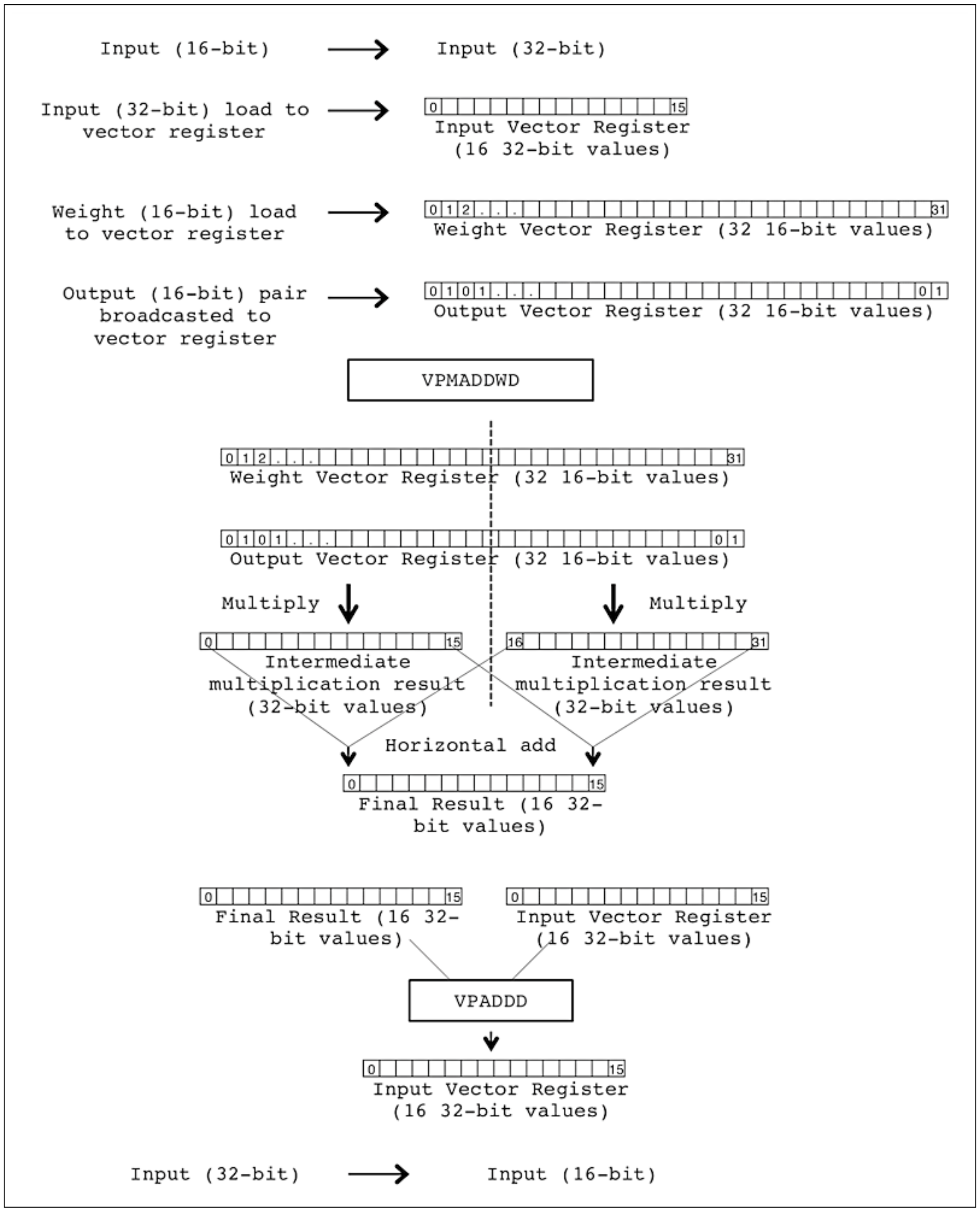
Input (16-bit)
Input (32-bit)
Input (32-bit) load to vector register
Input Vector Register (16 32-bit values)
Weight (16-bit) load to vector register
Weight Vector Register (32 16-bit values)
Output (16-bit) pair broadcasted to vector register
Output Vector Register (32 16-bit values)
VPMADDWD
Weight Vector Register (32 16-bit values)
Output Vector Register (32 16-bit values)
Multiply
Multiply
Intermediate multiplication result (32-bit values)
Intermediate multiplication result (32-bit values)
Horizontal add
Final Result (16 32-bit values)
Final Result (16 32-bit values)
Input Vector Register (16 32-bit values)
VPADDD
Input Vector Register (16 32-bit values)
Input (32-bit)
Input (16-bit)

Figure 3.11 : Low precision FMA strategy

```
for(ofm = 0; ofm < K; ++ofm) { //independent
 for(ifm = 0; ifm < C; ++ifm) { //independent
  for(img = 0; img < N; ++img) { //reduction
   for(oj = 0; oj < P; ++oj) {//reduction
  ij = oj * u;
  for(oi = 0; oi < Q; ++oi) {//reduction
   ii = oi * v;
   for(kj = 0; kj < R; ++kj) { //independent
    for(ki = 0; ki < S; ++ki) { //independent
     grad_weight(ofm, ifm, kj, ki) +=
    input(img, ifm, ij + kj, ii + ki) *
    grad_output(img, ofm, oj, oi);

} } } } } } }
```

Figure 3.12 : Pseudo code of naive direct convolution for weight update

3.5 Optimized Direct Convolution for Weight Update on KNL

For convenience of understanding, we reproduce Figure 3.12 here.

The optimizations that we applied to direct convolution in back propagation also applies here. So, we will only talk about the details that are different for weight update from back propagation.

3.5.1 Vectorization

For direct convolution in weight update, we vectorize along the blocking factor B_O of output feature map because weight tensor and output tensor both have B_O as their innermost dimension. So, we block the *ofm* loop by vector length *VLEN* and bring that loop to the innermost position and then vectorize. We do the fused multiply add operation using **vfmadd231ps** instruction as we did in section 3.4.1 with change in vector register arguments. As we are accumulating to weight tensor now, we use **zmm1** for weight tensor. We broadcast input tensor value to **zmm3** and use **zmm2** for output tensor. The resulting code is shown in Figure 3.13.

```
for(ofm1 = 0; ofm1 < K/VLEN; ++ofm1) { //blocked by VLEN
 for(ifm1 = 0; ifm1 < C/$B_I$; ++ifm1) {
  for(ifm2 = 0; ifm2 < $B_I$; ++ifm2) {
   for(img = 0; img < N; ++img) {
    for(oj = 0; oj < P; ++oj) {
     ij = oj * u;
     for(oi = 0; oi < Q; ++oi) {
      ii = oi * v;
      for(kj = 0; kj < R; ++kj) {
       for(ki = 0; ki < S; ++ki) {
        grad_weight_vec = vload(gard_weight[ofm1][ifm1][kj][ki][
            ifm2][0]);
        output_vec = vload(output[img][ofm1][oj][oi][0]);
        vfmadd231ps:: grad_weight_vec, grad_output_vec,
               broadcast(input[img][ifm1][ij+kj][ii+ki][ifm2]);
        vstore(gard_weight[ofm1][ifm1][kj][ki][ifm2][0],
            grad_weight_vec);
 } } } } } } } }
```

Figure 3.13 : Pseudo code for weight update after vectorization

3.5.2 Optimize for Temporal and Spatial Locality

We do a 2-D cache blocking along `oi` and `oj` loops in order to get temporal locality along weight tensor accesses. We register block on *ifm* loop and gain spatial locality for input tensor accesses and temporal locality for output tensor accesses. The pseudo code for weight update, after performing register blocking and cache blocking, is given in Figure 3.14.

3.5.3 Optimize loads and stores

To optimize load and stores of weight tensor, we hoist weight tensor loading and storing outside *oj_b* loop because it does not depend on output feature height, P, and output feature width, Q. We set up a 4-register ring buffer for output loads to load outputs in advance to their use in corresponding **fmadd**s; similar to what we do for weight loads in section 3.4.3. The resulting code is given in Figure 3.15.

```
for(ofm1 = 0; ofm1 < K/VLEN; ++ofm1) { //blocked by VLEN
 for(ifm1 = 0; ifm1 < C/$B_I$; ++ifm1) {
   for(img = 0; img < N; ++img) {
    for(oj = 0; oj < P/$B_P$; ++oj) {
     ij = oj * u;
     for(oi = 0; oi < Q/$B_Q$; ++oi) {
      ii = oi * v;
      for(kj = 0; kj < R; ++kj) {
       for(ki = 0; ki < S; ++ki) {
        for (oj_b = 0; oj_b < $B_P$; ++oj_b) { //unrolled
         for (oi_b = 0; oi_b < $B_Q$; ++oi_b) { //unrolled
           grad_weight_vec = vload(gard_weight[ofm1][ifm1][kj][ki
              ][0][0]);
           output_vec = vload(output[img][ofm1][oj*$B_P$+oj_b][oi*
              $B_Q$+oi_b][0]);
           vfmadd231ps:: grad_weight_vec, grad_output_vec,
              broadcast(input[img][ifm1][(oj*$B_P$+oj_b)*u+kj][(oi*
              $B_Q$+oi_b)*v+ki][0]);
           vstore(gard_weight[ofm1][ifm1][kj][ki][0][0],
              grad_weight_vec);

           grad_weight_vec = vload(gard_weight[ofm1][ifm1][kj][ki
              ][1][0]);
           output_vec = vload(output[img][ofm1][oj*$B_P$+oj_b][oi*
              $B_Q$+oi_b][0]);
           vfmadd231ps:: grad_weight_vec, grad_output_vec,
              broadcast(input[img][ifm1][(oj*$B_P$+oj_b)*u+kj][(oi*
              $B_Q$+oi_b)*v+ki][1]);
           vstore(gard_weight[ofm1][ifm1][kj][ki][1][0],
              grad_weight_vec);

           ...
 } } } } } } } } }
```

Figure 3.14 : Pseudo code for weight update after 2-D register blocking and cache blocking

```
for(ofm1 = 0; ofm1 < K/VLEN; ++ofm1) { //blocked by VLEN
 for(ifm1 = 0; ifm1 < C/$B_I$; ++ifm1) {
   for(img = 0; img < N; ++img) {
    for(oj = 0; oj < P/$B_P$; ++oj) {
     ij = oj * u;
     for(oi = 0; oi < Q/$B_Q$; ++oi) {
      ii = oi * v;
      for(kj = 0; kj < R; ++kj) {
       for(ki = 0; ki < S; ++ki) {
        grad_weight_vec_0 = vload(gard_weight[ofm1][ifm1][kj][ki
            ][0][0]);
        grad_weight_vec_1 = vload(gard_weight[ofm1][ifm1][kj][ki
            ][1][0]);
        ...
        for (oj_b = 0; oj_b < $B_P$; ++oj_b) { //unrolled
           output_vec_0 = vload(output[img][ofm1][oj*$B_P$+oj_b][oi*
               $B_Q$][0]);
           output_vec_1 = vload(output[img][ofm1][oj*$B_P$+oj_b][oi*
               $B_Q$+1][0]);
           output_vec_2 = vload(output[img][ofm1][oj*$B_P$+oj_b][oi*
               $B_Q$+2][0]);
           output_vec_3 = vload(output[img][ofm1][oj*$B_P$+oj_b][oi*
               $B_Q$+3][0]);

           vfmadd231ps:: grad_weight_vec_0, grad_output_vec_0,
               broadcast(input[img][ifm1][(oj*$B_P$+oj_b)*u+kj][(oi*
               $B_Q$)*v+ki][0]);
           vfmadd231ps:: grad_weight_vec_1, grad_output_vec_0,
               broadcast(input[img][ifm1][(oj*$B_P$+oj_b)*u+kj][(oi*
               $B_Q$)*v+ki][1]);
           ...

           output_vec_4 = vload(output[img][ofm1][oj*$B_P$+oj_b][oi*
               $B_Q$+4][0]);
           vfmadd231ps:: grad_weight_vec_0, grad_output_vec_1,
               broadcast(input[img][ifm1][(oj*$B_P$+oj_b)*u+kj][(oi*
               $B_Q$+1)*v+ki][0]);
           vfmadd231ps:: grad_weight_vec_1, grad_output_vec_1,
               broadcast(input[img][ifm1][(oj*$B_P$+oj_b)*u+kj][(oi*
               $B_Q$+1)*v+ki][1]);

           ...
        }
        vstore(gard_weight[ofm1][ifm1][kj][ki][0][0],
            grad_weight_vec_0);
        vstore(gard_weight[ofm1][ifm1][kj][ki][1][0],
            grad_weight_vec_1);
        ...
} } } } } } }
```

Figure 3.15 : Pseudo code for weight update after optimizing loads and stores

3.5.4 Runtime Code Specialization and Parallelization

As discussed in section 3.4.4, we use SMALLGEMM jus-in-time assembly generator [12] for runtime code specialization. In case of weight update, we take the code sequence inside `ki` loop in Figure 3.15 and abstract it out using SMALLGEMM JITer.

In Figure 3.15, the independent loops, which are easily parallelizable, are `ofm1`, `ifm1`, `kj`, and `ki`. Among these loops, `kj` and `ki` loops carry significant reuse for input tensor. So, we only parallelize `ofm1` and `ifm1` loops for thread-level parallelism on x86 architecture. However, sometimes this results in limited parallelism for some layers of popular CNNs. One solution is to try to parallelize `img` loop. But this loop is a reduction loop in weight update. So, we use thread local weight tensor buffer in this special case, for storing partial results. Then we do parallel reduction over these thread-local buffers to get the final results. Here, we pay a moderate cost in terms of memory requirement to gain more parallelism. The pseudo code with JIT interface and thread-level parallelism is given in Figure 3.16.

3.5.5 Software Prefetch

As discussed in section 3.4.7, we do software prefetch in JITed function to prefetch the data required for next JITed function invocation. In case of weight update, we need to prefetch in general $B_P * B_Q * B_I$ input elements, $B_P * B_Q * VLEN$ output elements, and $B_I * VLEN$ weight elements in a JITed function call. However, there are some corner cases. For example, we reuse the output elements across `ki` and `kj` loops. We do not need to prefetch output data for JITed function calls within these loops. Hence, we generate two variants of JITed code for weight update, one with output prefetch (*conv_wt_pf*) and another with no output prefetch (*conv_wt_pf_nooutput*). The pseudo code for weight update with software prefetching is given in Figure 3.17.

```
wt_desc = setup weight update descriptor using N,C,H,W,K,R,S,u,v,
    pad_h,pad_w;
conv_wt = wt_jit(wt_desc, ...); //Generate JIT code
if (enough parallelism) {
  temp_wt_ptr = &(weight[0][0][0][0][0][0]);
#pragma omp for collapse(2)
} else { //not enough parallelism
  temp_wt_ptr = &(thread_local_wt_tensor);
#pragma omp for collapse(3)
}
for(ofm1 = 0; ofm1 < K/VLEN; ++ofm1) {
 for(ifm1 = 0; ifm1 < C/$B_I$; ++ifm1) {
  for(img = 0; img < N; ++img) {
   for(oj = 0; oj < P/$B_P$; ++oj) {
    for(oi = 0; oi < Q/$B_Q$; ++oi) {
     for(kj = 0; kj < R; ++kj) {
      for(ki = 0; ki < S; ++ki) {
       const float *inp_ptr = &(grad_input[img][ifm1][oj*$B_P$*u+kj
           ][oi*$B_Q$*v+ki][0]);
       const float *out_ptr = &(grad_output[img][ofm1][oj*$B_P$][oi*
           $B_Q$][0]);
       float *wt_ptr = &(temp_wt_ptr[ofm1][ifm1][kj][ki][0][0]);
       conv_wt(input_ptr, wt_ptr, out_ptr);
} } } } } } }
```

Figure 3.16 : Pseudo code for weight update after runtime code specialization and parallelization

```
wt_desc = setup weight update descriptor using N,C,H,W,K,R,S,u,v,
    pad_h,pad_w;
conv_wt_pf = wt_jit(wt_desc, PREFETCH_ALL);
conv_wt_pf_nooutput = wt_jit(wt_desc, PREFETCH_NO_OUTPUT);
if (enough parallelism) {
  temp_wt_ptr = &(weight[0][0][0][0][0][0]);
#pragma omp for collapse(2)
} else { //not enough parallelism
  temp_wt_ptr = &(thread_local_wt_tensor);
#pragma omp for collapse(3)
}
for(ofm1 = 0; ofm1 < K/VLEN; ++ofm1) {
 for(ifm1 = 0; ifm1 < C/BI; ++ifm1) {
  for(img = 0; img < N; ++img) {
   for(oj = 0; oj < P/BP; ++oj) {
    for(oi = 0; oi < Q/BQ; ++oi) {
     for(kj = 0; kj < R; ++kj) {
      for(ki = 0; ki < R; ++ki) {
       const float *inp_ptr = &(grad_input[img][ifm1][oj*BP*u+kj]
                                                        [oi*BQ*v+ki
                                                           ][0]);
       const float *out_ptr = &(grad_output[img][ofm1][oj*BP][oi*
           BQ][0]);
       float *wt_ptr = &(temp_wt_ptr[ofm1][ifm1][kj][ki][0][0]);
       if (within kj loop) {
        conv_wt_pf_nooutput(input_ptr, wt_ptr, out_ptr,
               input_prefetch_ptr,
               weight_prefetch_ptr,
               NULL //No output prefetch);
       } else {
        conv_wt_pf(input_ptr, wt_ptr, out_ptr,
                   input_prefetch_ptr,
                   weight_prefetch_ptr,
                   output_prefetch_ptr);
       }
} } } } } } }
```

Figure 3.17 : Pseudo code for weight update with software prefetching

Chapter 4

Performance Results on KNL

We evaluated our optimized implementation of direct convolution for back propagation and weight update on Intel x86 platform named Knights Landing (KNL). We give our performance results on a single-socket Intel Xeon Phi 7250 processor with 68 cores, 1.2 GHz mesh-clock, 16 GB MCDRAM@7.2 GT, 96 GB DDR4-2400, FLAT memory mode and QUADRANT cluster mode. For details, one can look into [10]. In our experiments, we enabled Turbo mode, which makes the processor run at 1.3 GHz. We use the Intel®C++ Compiler (ICC) with "*-O2*" flag for the compilation. All the data were kept in the MCDRAM instead of DDR4 for higher bandwidth and latency. We achieve this by using"`numactl membind=1`". We use 64 threads for all the experiments.

For evaluation of performance across different state-of-the-art CNNs, we choose five popular CNNs - Alexnet [5], Overfeat [17], Vgga [18], GoogleNet_V1 [19], and DeepBench [20]. We select these topologies as each of them will stress several code generation aspects. The most important aspects are: a) the feature map dimensions of Overfeat and Alexnet are close or at the boundary of hardware latencies for FMA units b) odd number for the dimensions of images in Alexnet c) inner layers of Overfeat and Vgga are wide layers which exceed the L1 cache size, therefore cache data management and prefetching properly is a necessity for these cases d) Vgga layers have large feature map dimensions that require sufficient tiling even within the feature maps e) GoogleNet_V1 has a widely varying configurations from small feature map dimensions to large feature map dimensions, small feature height and width to

Table 4.1 : Parameters for Convolutional Layers of AlexNet, Overfeat, and Vgga

Layers	W	H	N	C	K	R	S	P	Q	Stride
Alexnet_CONV1	227	227	256	3	64	11	11	55	55	4
Alexnet_CONV2	27	27	256	64	192	5	5	27	27	1
Alexnet_CONV3	13	13	256	192	384	3	3	13	13	1
Alexnet_CONV4	13	13	256	384	256	3	3	13	13	1
Alexnet_CONV5	13	13	256	256	256	3	3	13	13	1
Overfeat_CONV1	231	231	256	3	96	11	11	56	56	4
Overfeat_CONV2	28	28	256	96	256	5	5	24	24	1
Overfeat_CONV3	12	12	256	256	512	3	3	12	12	1
Overfeat_CONV4	12	12	256	512	1024	3	3	12	12	1
Overfeat_CONV5	12	12	256	1024	1024	3	3	12	12	1
Vgga_CONV1	224	224	128	3	64	3	3	224	224	1
Vgga_CONV2	112	112	256	64	128	3	3	112	112	1
Vgga_CONV3	56	56	256	128	256	3	3	56	56	1
Vgga_CONV4	56	56	256	256	256	3	3	56	56	1
Vgga_CONV5	28	28	256	256	512	3	3	28	28	1
Vgga_CONV6	28	28	256	512	512	3	3	28	28	1
Vgga_CONV7	14	14	256	512	512	3	3	14	14	1
Vgga_CONV8	14	14	256	512	512	3	3	14	14	1

large feature height and width, small filter height and width to large filter height and width; in a sense it tests our implementation on the aspect of wide applicability, and finally f) DeepBench represent a wide range of corner cases which are good for testing performance on worst case configurations. The parameters of convolutional layers of AlexNet, Overfeat, and Vgga are represented in Table 4.1, GoogleNet_V1 in Table 4.2, and DeepBench in Table 4.3.

Table 4.2 : Parameters for Convolutional Layers of GoogleNetV1

Layers	W	H	N	C	K	R	S	P	Q	Stride
Googlenetv1_CONV1	224	224	128	3	64	7	7	112	112	2
Googlenetv1_CONV2	56	56	128	64	64	1	1	56	56	1
Googlenetv1_CONV3	56	56	128	64	192	3	3	56	56	1
Googlenetv1_CONV4	28	28	128	192	64	1	1	28	28	1
Googlenetv1_CONV5	28	28	128	192	96	1	1	28	28	1
Googlenetv1_CONV6	28	28	128	96	128	3	3	28	28	1
Googlenetv1_CONV7	28	28	128	192	16	1	1	28	28	1
Googlenetv1_CONV8	28	28	128	16	32	5	5	28	28	1
Googlenetv1_CONV9	28	28	128	192	32	1	1	28	28	1
Googlenetv1_CONV10	28	28	128	256	128	1	1	28	28	1
Googlenetv1_CONV11	28	28	128	128	192	3	3	28	28	1
Googlenetv1_CONV12	28	28	128	256	32	1	1	28	28	1
Googlenetv1_CONV13	28	28	128	32	96	5	5	28	28	1
Googlenetv1_CONV14	28	28	128	256	64	1	1	28	28	1
Googlenetv1_CONV15	14	14	128	480	192	1	1	14	14	1
Googlenetv1_CONV16	14	14	128	480	96	1	1	14	14	1
Googlenetv1_CONV17	14	14	128	96	208	3	3	14	14	1
Googlenetv1_CONV18	14	14	128	480	16	1	1	14	14	1
Googlenetv1_CONV19	14	14	128	16	48	5	5	14	14	1
Googlenetv1_CONV20	14	14	128	480	64	1	1	14	14	1
Googlenetv1_CONV21	14	14	128	512	160	1	1	14	14	1
Googlenetv1_CONV22	14	14	128	512	112	1	1	14	14	1

Continued on next page

Table 4.2 : Parameters for Convolutional Layers of GoogleNetV1

Layers	W	H	N	C	K	R	S	P	Q	Stride
Googlenetv1_CONV23	14	14	128	112	224	3	3	14	14	1
Googlenetv1_CONV24	14	14	128	512	32	1	1	14	14	1
Googlenetv1_CONV25	14	14	128	32	64	5	5	14	14	1
Googlenetv1_CONV26	14	14	128	512	64	1	1	14	14	1
Googlenetv1_CONV27	14	14	128	512	128	1	1	14	14	1
Googlenetv1_CONV28	14	14	128	128	256	3	3	14	14	1
Googlenetv1_CONV29	14	14	128	512	144	1	1	14	14	1
Googlenetv1_CONV30	14	14	128	144	288	3	3	14	14	1
Googlenetv1_CONV31	14	14	128	512	32	1	1	14	14	1
Googlenetv1_CONV32	14	14	128	32	64	5	5	14	14	1
Googlenetv1_CONV33	14	14	128	528	256	1	1	14	14	1
Googlenetv1_CONV34	14	14	128	528	160	1	1	14	14	1
Googlenetv1_CONV35	14	14	128	160	320	3	3	14	14	1
Googlenetv1_CONV36	14	14	128	528	32	1	1	14	14	1
Googlenetv1_CONV37	14	14	128	32	128	5	5	14	14	1
Googlenetv1_CONV38	14	14	128	528	128	1	1	14	14	1
Googlenetv1_CONV39	7	7	128	832	256	1	1	7	7	1
Googlenetv1_CONV40	7	7	128	832	160	1	1	7	7	1
Googlenetv1_CONV41	7	7	128	160	320	3	3	7	7	1
Googlenetv1_CONV42	7	7	128	832	32	1	1	7	7	1
Googlenetv1_CONV43	7	7	128	32	128	5	5	7	7	1
Googlenetv1_CONV44	7	7	128	832	128	1	1	7	7	1

Continued on next page

Table 4.2 : Parameters for Convolutional Layers of GoogleNetV1

Layers	W	H	N	C	K	R	S	P	Q	Stride
Googlenetv1_CONV45	7	7	128	832	384	1	1	7	7	1
Googlenetv1_CONV46	7	7	128	832	192	1	1	7	7	1
Googlenetv1_CONV47	7	7	128	192	384	3	3	7	7	1
Googlenetv1_CONV48	7	7	128	832	48	1	1	7	7	1
Googlenetv1_CONV49	7	7	128	48	128	5	5	7	7	1

Table 4.3 : Parameters for Convolutional Layers of DeepBench

Layers	W	H	N	C	K	R	S	P	Q	Stride
Deepbench_CONV1	700	161	4	1	32	20	5	79	341	2
Deepbench_CONV2	700	161	8	1	32	20	5	79	341	2
Deepbench_CONV3	700	161	16	1	32	20	5	79	341	2
Deepbench_CONV4	700	161	32	1	32	20	5	79	341	2
Deepbench_CONV5	341	79	4	32	32	10	5	38	166	2
Deepbench_CONV6	341	79	8	32	32	10	5	38	166	2
Deepbench_CONV7	341	79	16	32	32	10	5	38	166	2
Deepbench_CONV8	341	79	32	32	32	10	5	38	166	2
Deepbench_CONV9	480	48	16	1	16	3	3	48	480	1
Deepbench_CONV10	240	24	16	16	32	3	3	24	240	1
Deepbench_CONV11	120	12	16	32	64	3	3	12	120	1
Deepbench_CONV12	60	6	16	64	128	3	3	6	60	1
Deepbench_CONV13	108	108	8	3	64	3	3	108	108	1

Continued on next page

Table 4.3 : Parameters for Convolutional Layers of DeepBench

Layers	W	H	N	C	K	R	S	P	Q	Stride
Deepbench_CONV14	54	54	8	64	64	3	3	54	54	1
Deepbench_CONV15	27	27	8	128	128	3	3	27	27	1
Deepbench_CONV16	14	14	8	128	256	3	3	14	14	1
Deepbench_CONV17	7	7	8	256	512	3	3	7	7	1
Deepbench_CONV18	224	224	8	3	64	3	3	224	224	1
Deepbench_CONV19	112	112	8	64	128	3	3	112	112	1
Deepbench_CONV20	56	56	8	128	256	3	3	56	56	1
Deepbench_CONV21	28	28	8	256	512	3	3	28	28	1
Deepbench_CONV22	14	14	8	512	512	3	3	14	14	1
Deepbench_CONV23	7	7	8	512	512	3	3	7	7	1
Deepbench_CONV24	224	224	16	3	64	3	3	224	224	1
Deepbench_CONV25	112	112	16	64	128	3	3	112	112	1
Deepbench_CONV26	56	56	16	128	256	3	3	56	56	1
Deepbench_CONV27	28	28	16	256	512	3	3	28	28	1
Deepbench_CONV28	14	14	16	512	512	3	3	14	14	1
Deepbench_CONV29	7	7	16	512	512	3	3	7	7	1
Deepbench_CONV30	224	224	16	3	64	7	7	224	224	1
Deepbench_CONV31	28	28	16	192	32	5	5	28	28	1
Deepbench_CONV32	28	28	16	192	64	1	1	28	28	1
Deepbench_CONV33	14	14	16	512	48	5	5	14	14	1
Deepbench_CONV34	14	14	16	512	192	1	1	14	14	1
Deepbench_CONV35	7	7	16	832	256	1	1	7	7	1

Continued on next page

Table 4.3 : Parameters for Convolutional Layers of DeepBench

Layers	W	H	N	C	K	R	S	P	Q	Stride
Deepbench_CONV36	7	7	16	832	128	5	5	7	7	1

We measure the performance of our approach with three different data layouts (refer to section 3.3) 1) input, output, and weight tensors all are in our custom format, we call it Custom/Custom 2) input and output tensors is in TensorFlow format NHWC and weight tensor is in our Custom format, we call this setting NHWC/Custom 3) input and output is in TensorFlow format NHWC and also weight tensor is in TensorFlow format RSCK.

One thing to mention is that even though we implemented our direct convolution approach with low precision instructions as discussed in section 3.4.8 and also packed **`fmadd`** instructions as discussed in section 3.4.9, we can not provide performance results for these implementations at this point of time. This is because, to the best of our knowledge, current Intel x86 architectures do not have support for these instructions. But as the updated Intel x86 ISA includes these instructions, our projection is that the future Intel x86 architectures will have support for these instructions.

The performance results for Back Propagation are represented in the following way: a) results for AlexNet, Overfeat, and Vgga are in Figure 16, b) results for GoogleNetV1 are in Figure 17, and c) results for DeepBench are in Figure 18. Similarly, performance results for weight update are organized in the following way: a) results for AlexNet, Overfeat, and Vgga are in Figure 19, b) results for GoogleNetV1 are in Figure 20, and finally c) results for DeepBench are in Figure 21. For Back Propagation, we do report performance result for the layers for which we cannot perform vectorization and the execution fallbacks to nave implementation for producing correct

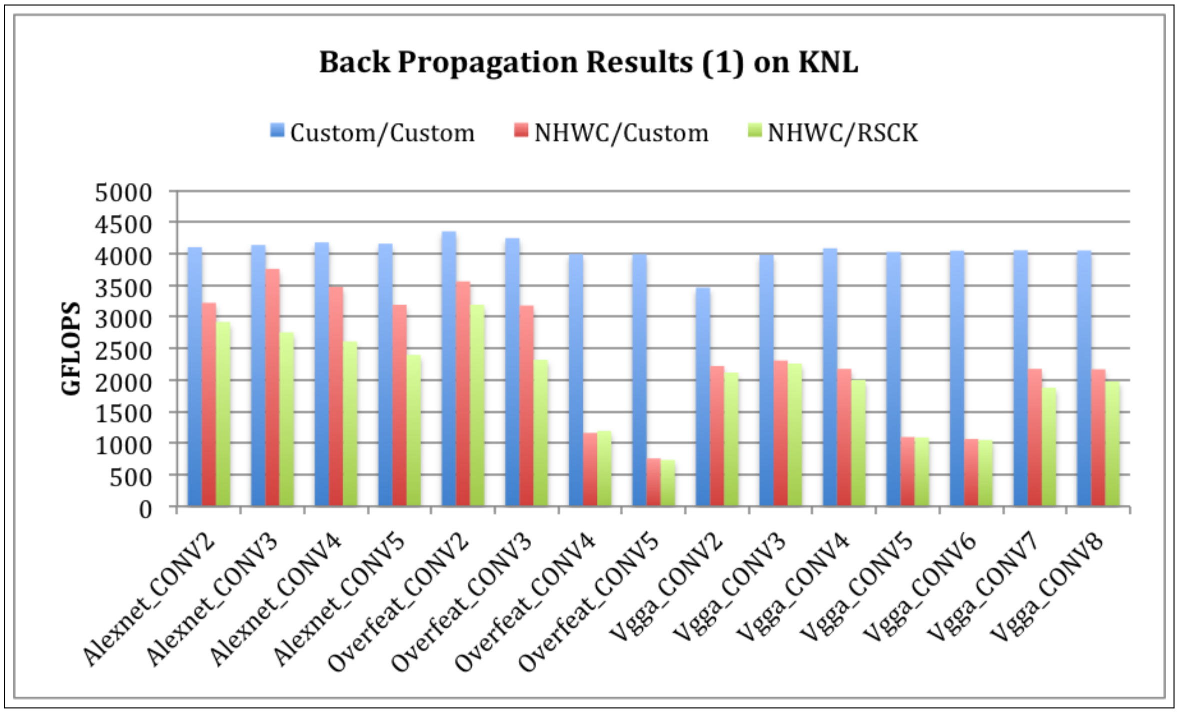

Figure 4.1 : Performance results for Back Propagation on AlexNet, Overfeat, and Vgga

results. As we observe from Figure 4.1, we consistently get 4-4.5 TeraFlops across the convolutional layers of AlexNet, Overfeat, and Vgga for our Custom/Custom data format. This is 80% of the theoretical peak (6 TFLOPS) on KNL. Please note, large HPC-style (Top500 benchmark [21]) matrix multiplications hit close to 80% of the machines peak. Hence, we are hitting the maximum possible peak performance on KNL. Due to a two-issue wide machine, this is the maximum achievable peak performance of the machine. However, there is serious degradation in performance when we use NHWC/Custom or NHWC/RSCK format. Our understanding behind this performance drop is 1) increased probability of conflict miss and 2) increased TLB pressure due to these data formats. As input feature map C and output feature map K are powers of 2 and our L1 data cache is 8-way set associative, the probability of conflict miss increases if we make C or K as the innermost dimension for input and output tensors respectively. The situation gets even worse when put K as the

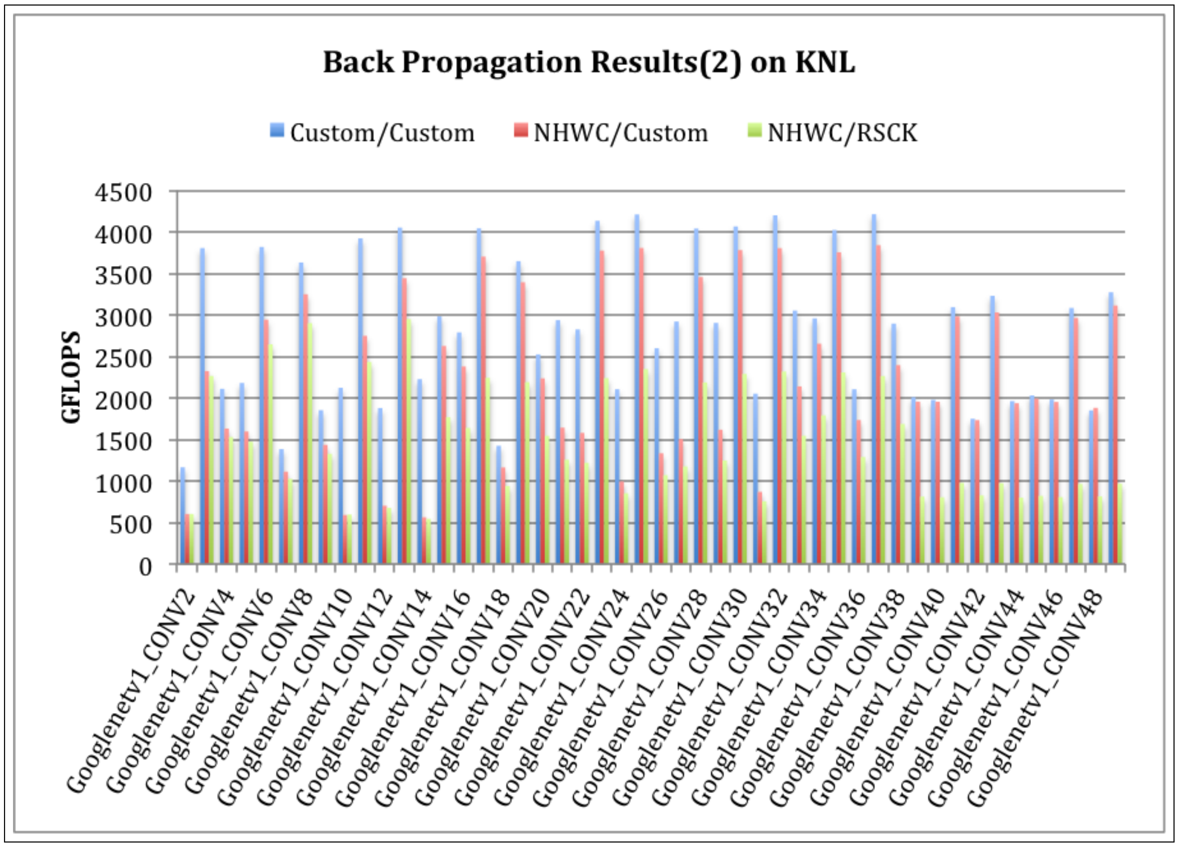

Figure 4.2 : Performance results for Back Propagation on GoogleNetV1

innermost dimension and C just after K, for weight tensor. Apart from increasing the probability of conflict miss, TLB pressure also increases with these data formats due to our tensor data access pattern. We do register blocking along W. If we put C and K as the innermost dimension and they have high values, which is typically the case, we might end up accessing completely different page tables for each index along W. Furthermore, we access weight tensor along R and S inside the JITed function. If the data format for weight is RSCK, we also might end up accessing completely different page tables for each access along R and S. This scenario increases TLB pressure tremendously, and the cost of TLB miss is very high on KNL.

From performance results for GoogleNetV1 in Figure 4.2, we see that we get excellent performance (around 4 TFLOPS) in Custom/Custom data format for 3x3 and

5x5 convolutions when W and H are more than 14. These are ideal scenarios for our implementation because we get good reuse of input tensor and the value of W is just perfect to hide the FMA latency on KNL. However, the performance drops to 3-3.5TFLOPS when W and H drop to 7. The reason is that we cannot hide the FMA latency completely in this case. We use extra accumulators to handle this scenario as discussed in section 3.4.6 but it has some overhead, which results in some performance drop. In case of 1x1 convolutions, we get around 2-3 TFLOPS. This significant drop in performance is observed because mainly there is no reuse of output and input data along R and S. This results in low FMA to loads/stores ratio and makes the performance depend more on memory bandwidth. For NHWC/Custom and NHWC/RSCK data formats, the performance is a less than what we achieve in Custom/Custom data format as expected.

In Figure 4.3, the performance from DeepBench shows how our implementation performs in corner cases. It still performs reasonably well (3.5-4TFLOPS) when the suitable conditions met, i.e. R and S are greater than 1, H and W are 14 or 28. In general, the performance on DeepBench suffers due to the reason that most of the layer has very small (less than 32) mini-batch size, N. In case of back propagation, we do a costly transpose of weight tensor as discussed in section 3.4.1. We amortize this cost over the mini-batch size because same weight tensor is used across all the images. So, when the number of images is not enough to amortize the cost of transposing weight tensor, we see a serious degradation in performance. Besides this major issue, some convolutional layers in DeepBench have values for C and K that do not yield enough parallelism for 64 threads. For example, layer 5, 6, 7, 8, and 10. This affects the performance severely.

As we see from Figure 4.4, 4.5, and 4.6, performance for weight update also follows the same trend of Custom/Custom data format having the highest performance

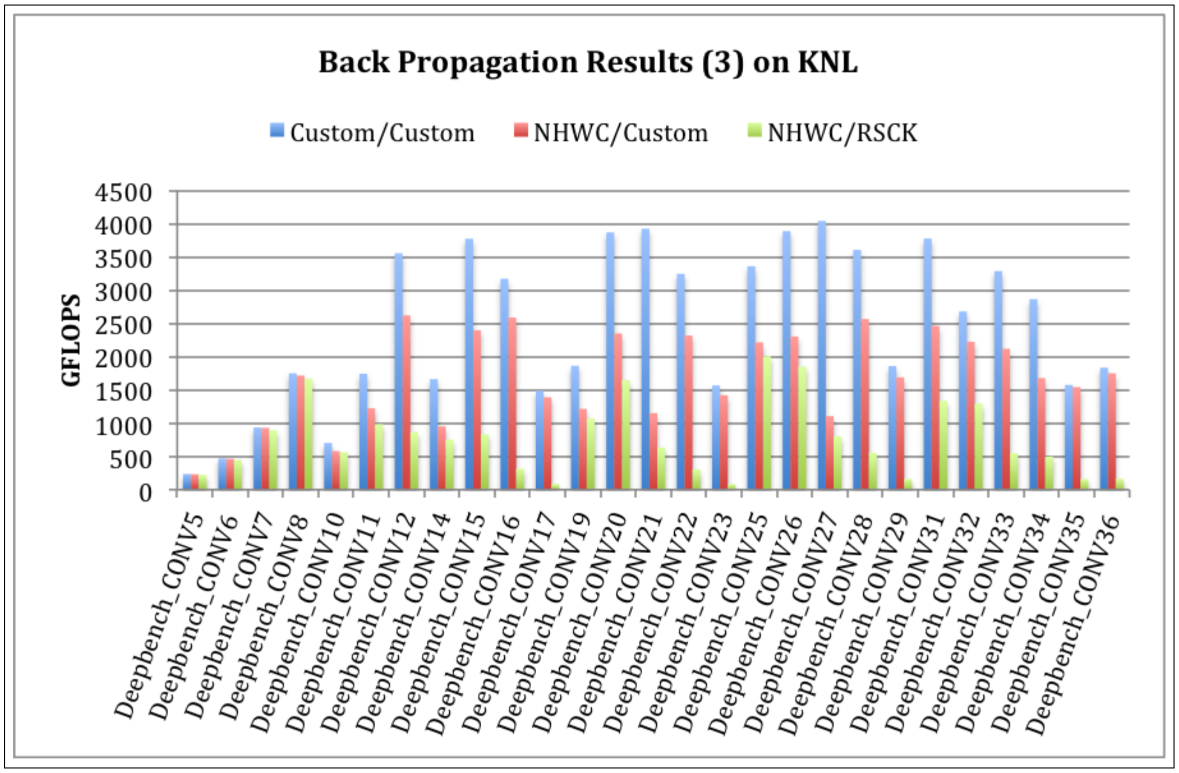

Figure 4.3 : Performance results for Back Propagation on DeepBench

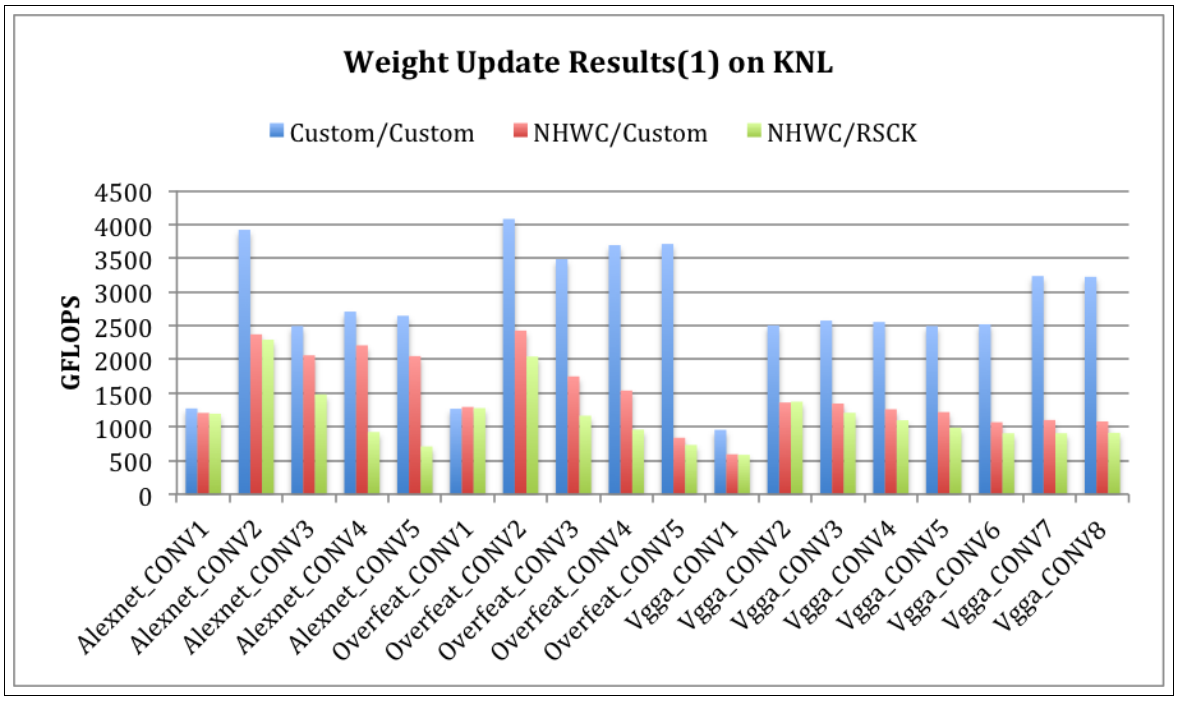

Figure 4.4 : Performance results for Weight Update on AlexNet, Overfeat, and Vgga

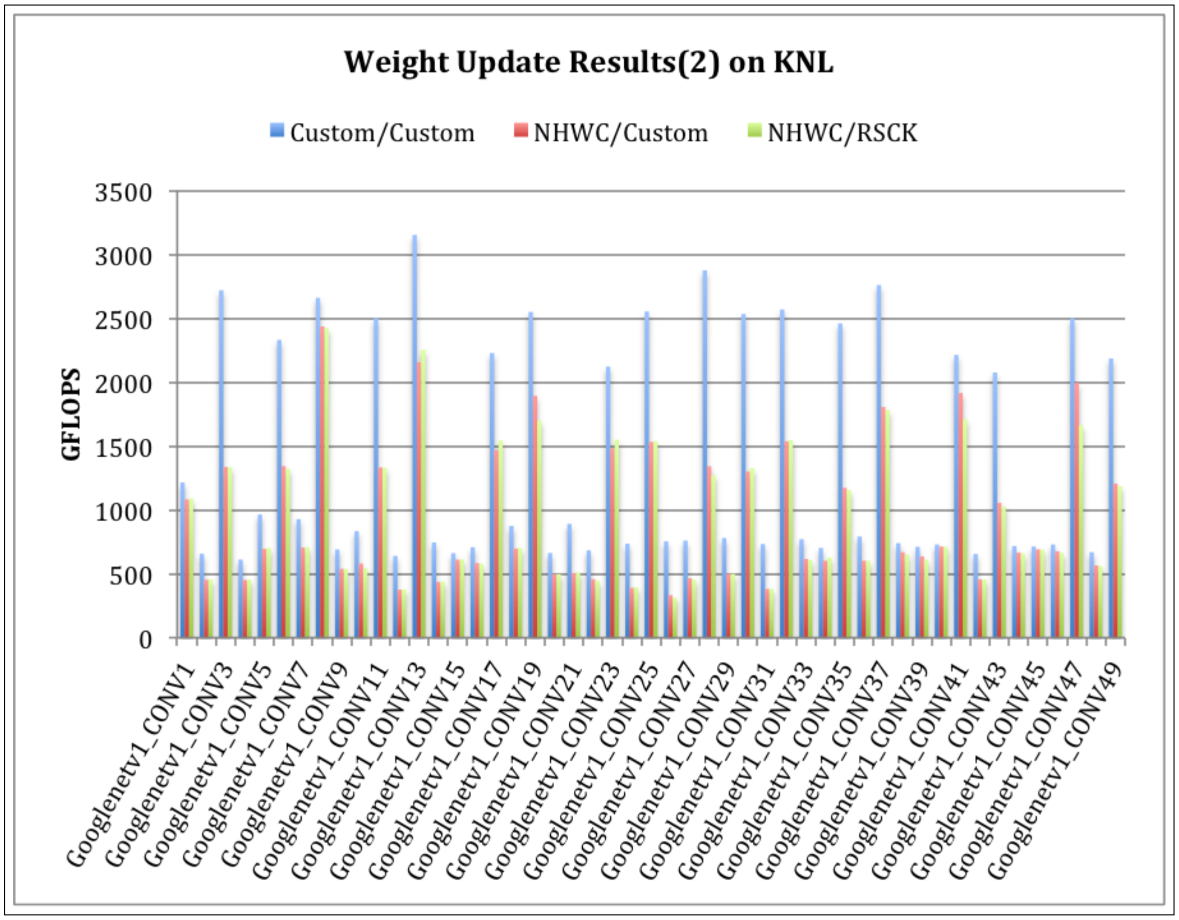

Figure 4.5 : Performance results for Weight Update on GoogleNetV1

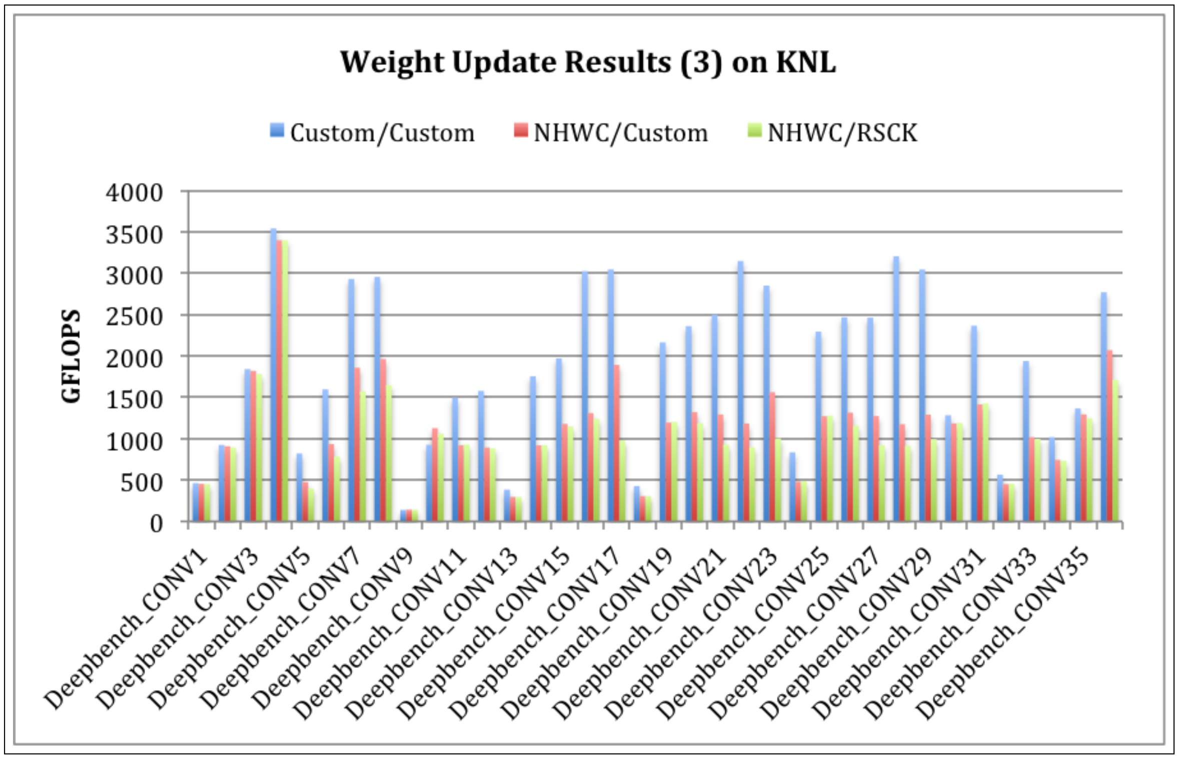

Figure 4.6 : Performance results for Weight Update on DeepBench

among all the data formats, followed by NHWC/Custom, and NHWC/RSCK gives the lowest performance. In case of AlexNet, Overfeat, and Vgga, we get 3-4TFLOPS as peak performance, for some layers we get 2.5 TFLOPS, and in worst cases, we get around 1TFLOPS. If we observe carefully, these worst cases are associated with first layers of the networks where ifm is 3. In this scenario, two adverse situation occurs 1) We do not have enough parallelism which results in using thread-local buffers 2) As we reduce across mini-batch size, N, the reduction work per weight element becomes quite high. We get to see the impact of the second factor more clearly on GoogleNetV1 and DeepBench. On GoogleNetV1, the factor causes havoc on the performance for all the convolutional layers having R=1, S=1. In these cases, it becomes a giant reduction problem which is memory bandwidth bound. On DeepBench, we see good performance (around 3 TFLOPS) when N, H, W (these are the dimensions

Table 4.4 : Back Propagation: Base Line Comparison

CNN Layers	Our Approach (GFLOPs)	ICC with -O3 (GFLOPs)	Improvement Factor
Alexnet_CONV2	4102.5	19.036	215.51x
Alexnet_CONV3	4136.2	10.418	397.02x
Alexnet_CONV4	4176.8	10.413	401.11x
Alexnet_CONV5	4156.9	10.403	399.59x
Overfeat_CONV2	4352	8.3448	521.52x
Overfeat_CONV3	4245.7	5.7789	734.69x
Overfeat_CONV4	3993.4	6.8257	585.05x
Overfeat_CONV5	3990.1	5.1609	773.14x
Vgga_CONV2	3462.3	8.6424	400.62x
Vgga_CONV3	3981.4	7.5249	529.1x
Vgga_CONV4	4084.9	8.6536	472.05x
Vgga_CONV5	4027.4	7.7366	520.56x
Vgga_CONV6	4045.4	7.6822	526.59x
Vgga_CONV7	4053	8.4561	479.3x
Vgga_CONV8	4049.6	5.7651	702.43x

over which reduction occurs) is low (N16, H and W 14) and C and K are sufficiently high to have enough parallelism for 64 threads. For example, convolutional layers 16 and 17 of DeepBench have these parameters.

We also perform a base line comparison of our JIT-based approach with naive C codes compiled with ICC while turning on `-O3` optimization flag, i.e. we do not perform any optimization and leave all to the standard compiler. We used `-O3` flag because `-O3` optimization flag specifically enables vectorization and software prefetching. We present the performance comparison for back propagation in Table 4.4 and weight update in Table 4.5. The results are for convolution layers from AlexNet, Overfeat, and Vgga. As can be seen from the tables, we achieve up to 700x performance improvement for back propagation and up to 500x for weight update. This emphasizes the need for careful optimization to achieve good performance on KNL architecture.

Table 4.5 : Weight Update: Base Line Comparison

CNN Layers	Our Approach (GFLOPs)	ICC with -O3 (GFLOPs)	Improvement Factor
Alexnet_CONV1	1272.2	34.959	36.39x
Alexnet_CONV2	3920.7	18.492	212.02x
Alexnet_CONV3	2494.6	10.034	248.61x
Alexnet_CONV4	2708	10.008	270.58x
Alexnet_CONV5	2646.2	9.8914	267.53x
Overfeat_CONV1	1267.2	10.394	121.92x
Overfeat_CONV2	4082.1	8.6243	473.33x
Overfeat_CONV3	3486.3	5.8512	595.83x
Overfeat_CONV4	3693.4	6.7614	546.25x
Overfeat_CONV5	3710.9	6.9178	536.43x
Vgga_CONV1	955.11	3.8885	245.62x
Vgga_CONV2	2504	6.9011	362.84x
Vgga_CONV3	2575	6.8821	374.16x
Vgga_CONV4	2553.8	5.4316	470.17x
Vgga_CONV5	2494.2	8.7221	285.96x
Vgga_CONV6	2520.1	8.5131	296.03x
Vgga_CONV7	3235.4	6.6083	489.6x
Vgga_CONV8	3224	6.1214	526.68x

Table 4.6 : Back Propagation: GEMM Method Comparison

CNN Layers	Our Approach (GFLOPs)	GEMM Method with -O3 (GFLOPs)	Improvement Factor
Alexnet_CONV2	4102.5	125.42	32.71x
Alexnet_CONV3	4136.2	261.72	15.8x
Alexnet_CONV4	4176.8	355.53	11.75x
Alexnet_CONV5	4156.9	282.5	14.71x
Overfeat_CONV2	4352	179.26	24.28x
Overfeat_CONV3	4245.7	291.72	14.55x
Overfeat_CONV4	3993.4	444.94	8.98x
Overfeat_CONV5	3990.1	578.66	6.9x
Vgga_CONV2	3462.3	128.6	26.92x
Vgga_CONV3	3981.4	236.54	16.83x
Vgga_CONV4	4084.9	385.46	10.6x
Vgga_CONV5	4027.4	381.82	10.55x
Vgga_CONV6	4045.4	380.27	10.64x
Vgga_CONV7	4053	584.62	6.93x
Vgga_CONV8	4049.6	438.54	9.23x

Furthermore, we make a performance comparison of our JIT-based approach of direct convolution with GEMM formulation of convolution. In case of GEMM formulation, we perform the GEMM computation with Intel MKL single precision BLAS routines. We compile the code for GEMM formulation with Intel ICC compiler while turning on `-O3` optimization flag. The performance comparison results are over convolution layers of AlexNet, Overfeat, and Vgga. Table 4.6 shows the results for back propagation and Table 4.7 lists the results for weight update. We would like to mention that we omitted the convolution layers where memory footprint was larger than the size of on-chip high bandwidth MCDRAM of KNL. As can be seen from the results, we achieve 7x to 30x speed up for back propagation and 18x to 60x speed up for weight update. These results support the fact that image flattening operation (`im2col`) does have a significant performance penalty.

Table 4.7 : Weight Update: GEMM Method Comparison

CNN Layers	Our Approach (GFLOPs)	GEMM Method with -O3 (GFLOPs)	Improvement Factor
Alexnet_CONV2	3920.7	101.77	38.53x
Alexnet_CONV3	2494.6	78.709	31.69x
Alexnet_CONV4	2708	86.041	31.47x
Alexnet_CONV5	2646.2	84.775	31.21x
Overfeat_CONV2	4082.1	113.41	35.99x
Overfeat_CONV3	3486.3	72.76	47.92x
Overfeat_CONV4	3693.4	75.014	49.24x
Overfeat_CONV5	3710.9	63.372	58.56x
Vgga_CONV2	2504	122.05	20.52x
Vgga_CONV3	2575	138.13	18.64x
Vgga_CONV4	2553.8	139.29	18.33x
Vgga_CONV5	2494.2	132.79	18.78x
Vgga_CONV6	2520.1	133.11	18.93x
Vgga_CONV7	3235.4	133.57	24.22x
Vgga_CONV8	3224	81.161	39.72x

Chapter 5

Related Work

There exists a variety of high-level deep learning frameworks with the goal of improving the productivity of data scientists by abstracting away implementation details. CNTK [6], THEANO [7], TensorFlow [3], Torch [8], and Caffe [4] are the popular ones. They more or less represent neural networks as computation graphs where each node represents some operation on n-dimensional arrays or tensors. So, a convolution layer is represented as a node with convolution operation over n-dimensional tensor. When aiming for performance, most of these frameworks resort to library based approaches, i.e., they use cuDNN [13] on GPUs and use Intel Math Kernel Library [22] on CPUs under the hood. Our runtime code specialization techniques for convolution layers are complementary to the high-level frameworks and can be easily integrated into them.

5.1 GEMM Based Approach

There are two kinds of parallelism when performing convolutions in CNNs with a GEMM based approach. One is to perform the GEMM for the whole mini-batch using widely available parallel-GEMM strategies. The other one is to perform GEMM for each image in parallel. The advantage of the first approach is that we have highly optimized parallel-GEMM implementations readily available to be used in our applications. So, it saves a lot of effort in optimizing GEMM part of the computation. For this reason, deep learning frameworks such as THEANO, TensorFlow, Torch, and Caffe uses the first parallelization strategy. Interestingly, different scheduling of the computation can fetch better performance for the parallel-GEMM approach. Caffe con Troll [23] improves the performance of Caffe by executing a batch of image partitions instead of one image partition per core.

5.2 Convolution on GPUs

The cuDNN library [13] is the de facto standard for accelerating deep learning on NVIDIA GPUs. cuDNN provides efficient implementations of various layers found in deep neural networks including convolution layers. To the best of our knowledge, cuDNN mostly uses matrix-multiply(GEMM) formulation but have also integrated Winograd implementations [24] for small filter sizes in the recent releases. In the cuDNN paper [13], the authors state that it is difficult to achieve performance using direct convolution across a wide range of parameters and is also not portable across GPU generations. We debunk this claim for CPUs and show that direct convolution can yield peak performance. Apart from GEMM or direct convolution approach, efficient FFT based strategy for large filter sizes and non-unit stride convolutions has also been proposed for GPU [25].

5.3 Convolution on Other Hardware Accelerators

Apart from GPU, there have also been some efforts on accelerating convolution in CNNs using FPGAs [26, 27, 28] and ASICs [29, 30]. However, most of these works focus only on forward propagation phase of CNN computation.

5.4 Convolution on CPUs

Although GPUs are the most prevalent architecture used for deep learning, x86 CPU implementations are on the rise, such as MKL-DNN [31], NNPACK [32], and ZNN [33]. To the best of our knowledge, NNPACK and ZNN use FFT-based approaches and do not support Xeon Phi systems. We used direct convolution approach and showed how we could achieve peak performance on Xeon Phi.

5.5 JIT Based Approach

Recently, TensorFlow XLA [34] introduced a JIT compiler for TensorFlow which enables aggressive fusion at XLA high-level optimizer. Although Google in its Developer Summit 2017 [35] showed that they gained significant performance improvement on GPUs by enabling fusion at runtime, their CPU performance degraded by up to -600%. This indicates that achieving high performance on x86 architecture is more challenging and necessitates a more carefully designed fusion strategy to harness performance improvement through data reuse. This might be an interesting area of future work.

5.6 Distributed CNN Training

Due to the large amount of computation needed to perform training on CNNs, there are several works which use a cluster of distributed systems with multi-core CPUs. For example, Google's DistBelief [36], Microsoft Project Adam [37], and SINGA [38] exploit massively parallel distributed systems through data parallelism. There are two core challenges here - scheduling the computation across different nodes and distributed update of model parameters. The general strategy is to use several worker nodes to do the training in parallel on different subsets of training data and periodically synchronize model parameters between different nodes. Our work focuses on improving the performance of CNN training on a single node. So, our effort would improve the throughput of each worker node, and therefore our work may lead to performance improvement in distributed systems.

Chapter 6

Conclusion and Future Work

Convolution Neural Networks (CNN) are state-of-the-art Deep Neural Networks for image recognition applications today. The core of these CNNs is convolution layer, which performs a large number of small convolutions with irregular dimensions. CNNs require massive computing power, and it turns out that convolution operation is the key performance enabler for CNNs. Although training and inference, both are time-consuming, training, in particular, can take a huge amount of time, extending up to few weeks. The critical bottleneck in training step is back propagation and weight update stages which are hard to optimize. So, in this work, we try to address this challenging problem by optimizing convolutions in back propagation and weight update. Our strategy involves - 1) producing highly optimized code for the innermost loops at runtime using JIT and 2) parallelizing outermost loops using widely available threading libraries.

The goal of the first step is to produce vectorized code for the innermost loops while avoiding branching as much as possible because branch misprediction is very costly on Intel Knights Landing. As we are addressing only direct convolution, it is possible for us to make our JIT very lightweight and efficient using domain knowledge and reducing generality of application. For example, we use our domain knowledge to decide which dimension is a good candidate for vectorization. We also use our domain knowledge in the heuristics used for deciding unroll factor and register blocking factor from the parameter values at runtime. As our JIT is very fast, it adds virtually zero overhead in execution time, and the optimized code generated by JIT is utilized by the outermost loops several times. At runtime, our JIT generates efficient scheduling of vector loads and stores along with vector fused-multiply-add instructions for x86 architectures with AVX-512 support. It also handles software prefetching, which is an important factor for gaining good performance on Intel Knights Landing. Regarding optimization, we applied standard compiler optimizations, such as register blocking, cache blocking, and loop unrolling. We also added code generation for low precision data (16-bit and 8-bit) in our JIT.

The objective of the second step is to separate out fine grain parallelism and low-level optimization details from coarse grain parallelization strategy. It enables us to track thread-level parallelism easily. Other than code optimizations, we considered different data layouts for our input, output, and weight tensors. We give a custom data layout for our approach which seems to be fitting our computation pattern. Along with this, we also experimented with TensorFlow data layout.

We demonstrate that efficient implementation of direct convolution for back propagation and weight update with runtime code specialization can achieve close to peak performance on Intel Knights Landing processor. Furthermore, we analyze the performance results for state-of-the-art CNNs and gives insight into what brings the performance and what hinders. We have also shown and discussed the impact of different data layouts on performance. In future, we would like to investigate how much performance improvement can be achieved by tuning the runtime code generator parameters with an auto-tuner. Also, another interesting topic of future interest would be looking into performance for Winograd convolutions using our JIT-based approach.

Bibliography

[1] "Convolution." `https://developer.apple.com/library/content/documentation/Performance/Conceptual/vImgae/ConvolutionOperations/ConvolutionOperations.html`.

[2] "Intel Second Generation XeonPhi(Knights Landing KNL) Architecture." `https://csdl-images.computer.org/mags/mi/2016/02/figures/mmi20160200341.gif`.

[3] Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Vigas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[4] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, ACM, 2014.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep

convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[6] D. Yu, A. Eversole, M. Seltzer, K. Yao, O. Kuchaiev, Y. Zhang, F. Seide, Z. Huang, B. Guenter, H. Wang, J. Droppo, G. Zweig, C. Rossbach, J. Gao, A. Stolcke, J. Currey, M. Slaney, G. Chen, A. Agarwal, C. Basoglu, M. Padmilac, A. Kamenev, V. Ivanov, S. Cypher, H. Parthasarathi, B. Mitra, B. Peng, and X. Huang, "An introduction to computational networks and the computational network toolkit," tech. rep., Microsoft Research, October 2014. Microsoft Technical Report MSR-TR-2014–112.

[7] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python," in *Proc. 9th Python in Science Conf*, pp. 1–7, 2010.

[8] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[9] N. Systems, "NEON." `https://github.com/NervanaSystems/neon`, 2016.

[10] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *Ieee micro*, vol. 36, no. 2, pp. 34–46, 2016.

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[12] A. Heinecke, H. Pabst, and G. Henry, "Libxsmm: A high performance library for small matrix multiplications," *Poster and Extended Abstract Presented at SC*, 2015.

[13] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[14] *Intel®64 and IA-32 Architectures Software Developers Manual, Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z.* `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf`.

[15] *Intel®64 and IA-32 Architectures Software Developers Manual, March 2017.* `https://software.intel.com/sites/default/files/managed/3e/79/252046-sdm-change-document.pdf`.

[16] *Intel®Architecture Instruction Set Extensions Programming Reference.* `https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf`.

[17] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[20] "DeepBench." `https://svail.github.io/DeepBench`.

[21] H. Meur, E. Strohmaier, J. Dongarra, and H. Simon, "Top500 list." `https://www.top500.org/`, June 2016.

[22] Intel Corporation, Santa Clara, USA, *Intel Math Kernel Library Reference Manual*, 2009. ISBN 630813-054US.

[23] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré, "Caffe con troll: Shallow ideas to speed up deep learning," in *Proceedings of the Fourth Workshop on Data Analytics in the Cloud*, DanaC'15, (New York, NY, USA), pp. 2:1–2:4, ACM, 2015.

[24] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.

[25] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *arXiv preprint arXiv:1312.5851*, 2013.

[26] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 32–37, IEEE, 2009.

[27] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[28] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.

[29] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 269–284, ACM, 2014.

[30] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE Computer Society, 2014.

[31] "MKL-DNN." `https://github.com/1org/mkl-dnn`.

[32] M. Dukhan, "NNPACK." `https://github.com/Maratyszcza/NNPACK`, 2016.

[33] A. Zlateski, K. Lee, and H. S. Seung, "Znn–a fast and scalable algorithm for training 3d convolutional networks on multi-core and many-core shared memory machines," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pp. 801–811, IEEE, 2016.

[34] Google, "TensorFlow XLA." `https://www.tensorflow.org/performance/xla/`, 2016.

[35] Google, "TensorFlow Developer Summit." `https://events.withgoogle.com/tensorflow-dev-summit/`, 2017.

[36] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, pp. 1223–1231, 2012.

[37] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 571–582, USENIX Association, 2014.

[38] B. C. Ooi, K.-L. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. Tung, Y. Wang, *et al.*, "Singa: A distributed deep learning platform," in *Proceedings of the 23rd ACM international conference on Multimedia*, pp. 685–688, ACM, 2015.