FALCON LIBRARY: FAST IMAGE CONVOLUTION IN NEURAL NETWORKS ON INTEL ARCHITECTURE

Sangamesh Ragate, Andrey Vladimirov, Bonan Zhang

Colfax International

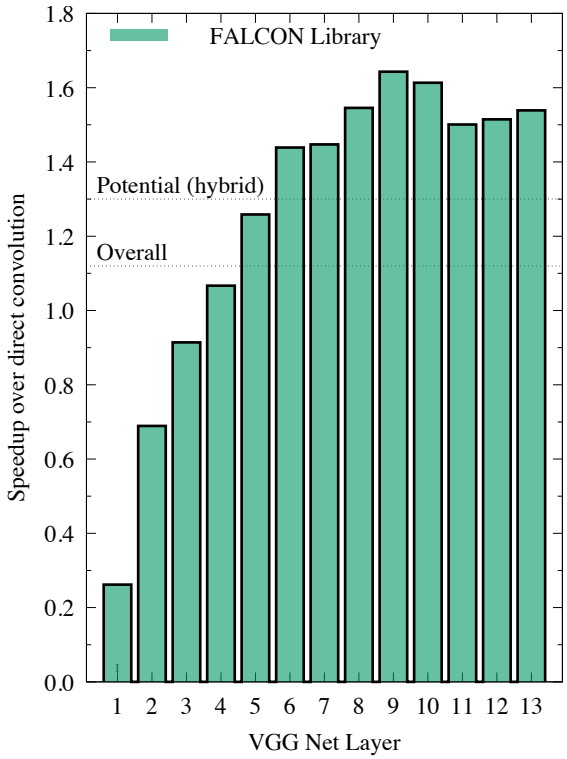
November 9, 2016

Abstract

We describe FALCON, an original open-source implementation of image convolution with a 3×3 filter based on Winograd's minimal filtering algorithm. Compared to direct convolution, Winograd's algorithm reduces the number of arithmetic operations at the cost of complicating the memory access pattern. This study is carried out in the context of image analysis in convolutional neural networks.

Our implementation combines C language code with BLAS function calls for general matrix-matrix multiplication. The code is optimized for Intel Xeon Phi processors x200 (formerly Knights Landing) with Intel Math Kernel Library (MKL) used for BLAS call to the SGEMM function.

To test the performance of FALCON in the context of machine learning, we benchmarked it for a set of image and filter sizes corresponding to the VGG Net architecture. In this test, FALCON achieves 10% greater overall performance than convolution from DNN primitives in Intel MKL. However, for some layers, FALCON is faster than MKL by 1.5x, but for other layers slower by as much as 4x. This indicates a possibility of a hybrid implementation with fast and direct convolution for a 30% speedup. High-bandwidth memory (MCDRAM) in Intel Xeon Phi x200 product family is a significant factor in the efficiency of the fast convolution algorithm.



Colfax International is a leading provider of high-performance computing solutions and expert-level educational programs for parallel computing. Ready-to-go Colfax systems include workstations, servers, clusters, storage and personal supercomputing solutions. Educational programs provided by Colfax enable software developers to achieve top performance on cutting-edge computing platforms, closing the loop between hardware innovation and progress in computational disciplines. The comprehensive set of services provided by Colfax delivers to its clients significant price/performance advantages and increased IT agility, which accelerates their business outcomes and paves the path to discovery. Colfax International's extensive customer base includes Fortune 1000 companies, educational institutions, and government agencies. Founded in 1987, Colfax International is based in Sunnyvale, California and is privately held.

1. Convolution in Machine Learning

Applications of deep neural networks (DNNs) to machine learning are diverse and promptly emerging, reaching the fields of assistive technologies, commerce, fundamental sciences, medical imaging and security (see, e.g., [1]). DNNs thrive with abundant data. As a consequence, training DNNs often requires expensive development time and powerful computing resources. Therefore, even small improvements in the efficiency of the fundamental building blocks of DNNs can benefit the field of machine learning.

In image analysis with DNNs, one building block has gained particular importance in recent years: the operation of convolution of images with a filter. This operation is used in convolutional DNNs (ConvNets), which rely on the mathematical operation of convolution for position-independent object identification in images [2].

Numerically, convolution may be performed directly. This method is expensive in terms of computational complexity. For an image of size $H \times W$ and filter of size $R \times S$, direct convolution requires $O(HWRS)$ operations. However, arithmetic operations in direct convolution can easily be collapsed to form the general matrix-matrix multiplication (GEMM) pattern [3]. This simplifies the design of convolution functions because the complexity of memory and cache traffic management is delegated to the implementation of GEMM. Efficient GEMM code exists in Basic Linear Algebra Subroutine (BLAS) libraries for nearly every computer architecture. In the case of Intel architecture, Intel Math Kernel Library (MKL) has highly efficient implementation of GEMM and of direct convolution expressed with matrix-matrix multiplication [4].

At the same time, it is possible to compute convolution with alternative methods that perform fewer arithmetic operations than the direct method. For example, fast Fourier transform (FFT) may be used to compute image convolution with complexity $O\left(HW \log(HW)\right)$ [5]. The asymptotic behavior of this algorithm predicts fewer operations than in direct method only if the filter is large enough: $RS \gg \log(HW)$. However, this approach is not useful for ConvNets because they typically use filters as small as 2×2 or 3×3 pixels. In this range, the performance of the FFT method is poor compared to the direct method. In the domain of small filters, Winograd's minimal filtering algorithm may be a better choice [6, 7]. This approach has the same asymptotic complexity as the direct method, $O(HWRS)$, but it reduces the number of operations by a constant factor. In this paper we present the implementation of convolution based on Winograd's minimal filtering algorithm for a filter size $R = S = 3$.

From here on, we refer to the convolution algorithm based on Winograd's algorithm as "fast convolution". This term, chosen by analogy with *fast* Fourier transform, signifies the algorithm performs fewer floating-point operations than the direct approach. At the same time, it is not trivial to implement a computer program performing "fast" convolution in less time than the direct method. This is because the fast algorithm requires data transformation with a complex memory access pattern, making it more difficult to express efficiently in code. The choice between an expensive yet simple algorithm (direct convolution) and less expensive but complicated algorithm (fast convolution) is not straightforward. It is difficult to predict, for instance, how well the hierarchy of memory and caches is able to serve the complex data access pattern of an algorithm based on strided or indirect memory access.

2. Winograd's Minimal FIR Filtering

The original Winograd's algorithm is applied to the computation of finite impulse response (FIR) filters. Direct application of 2 consecutive steps of a 3-tap FIR filter with coefficients g_i to a set of 4 elements d_i requires 6 additions and 6 multiplications:

$$F_0 = g_0 d_0 + g_1 d_1 + g_2 d_2, \quad (1)$$

$$F_1 = g_0 d_1 + g_1 d_2 + g_2 d_3. \quad (2)$$

The idea of Winograd's method is to compute these two filter outputs as

$$m_1 = (d_0 - d_2) g_0, \quad (3)$$

$$m_2 = (d_1 + d_2) \frac{g_0 + g_1 + g_2}{2}, \quad (4)$$

$$m_3 = (d_1 - d_3) g_2, \quad (5)$$

$$m_4 = (d_2 - d_1) \frac{g_0 - g_1 + g_2}{2}, \quad (6)$$

$$F_0 = m_1 + m_2 + m_3, \quad (7)$$

$$F_1 = m_2 + m_3 - m_4. \quad (8)$$

If we precompute the expressions $(g_0 + g_1 + g_2)/2$ and $(g_0 - g_1 + g_2)/2$, then this procedure requires 8 additions and 4 multiplications, which is equal to number of floating point operations in the direct method. However, if our goal is to apply multiple filters g_i to the same data d_i, then we can also precompute $(d_0 - d_2)$, $(d_1 + d_2)$, $(d_1 - d_3)$ and $(d_2 - d_1)$. With this done, calculations (3) - (8) would only require 4 additions and 4 multiplications, yielding a speedup of $(6 + 6)/(4 + 4) = 1.5$.

3. APPLICATION TO CONVNETS

In the context of ConvNets, the operation of convolution applies a total of F filters of size $R \times S$ to a batch of N images of size $H \times W$ with C channels in each. We enumerate filters with f, and channels with c. Each image we split into $T = (H - 2) \times (W - 2)/4$ tiles and enumerate these tiles within the image with t, which ranges from 0 to T. The images within a batch are enumerated with n, which ranges from 0 to N.

Direct convolution of a 3×3 filter $g_{f,c}$ with a 4×4 image $d_{c,t}$ to generate a 2×2 output tile $Y_{n,t,f,c}$ requires $3 \times 3 \times 2 \times 2 = 36$ multiplications and 36 additions. As shown in [7], Winograd's fast FIR filter computation can be generalized to 2D filters, which are mathematically similar to convolution. The fast method can be expressed as shown below:

$$Y_{n,t,f,c} = A^T \left[\left(B^T d_{n,t,c} B \right) \odot \left(G^T g_{c,f} G \right) \right] A, \quad (9)$$

where $d_{n,t,c}$ is a 4×4 matrix representing the image tile, $g_{c,f}$ is a 3×3 matrix representing channel c of filter f, $Y_{n,t,f,c}$ is a 2×2 matrix with the output of the convolution of $d_{n,t,c}$ with $g_{c,f}$,

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \quad (10)$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}, \quad (11)$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}. \quad (12)$$

In Equation (9) $\odot$ indicates element-wise multiplication. Assuming that we have many image tiles and multiple filters, we can precompute transformed image tiles $U_{n,t,c} \equiv \left(B^T d_{n,t,c} B \right)$ and transformed filters $V_{c,f} \equiv \left(G^T g_{c,f} G \right)$. With that done, the algorithm requires $4 \times 4 = 16$ multiplications between the transformed input and filters. For inverse transformations, 24 additions are required. This is already $(36 + 36)/(16 + 24) = 1.8$ times fewer operations than in the direct method. In addition to this, for the purposes of ConvNets, convolution must be applied to C image channels, and results for individual channels must be summed:

$$Y_{t,f} \equiv \sum_c Y_{n,t,f,c} = \sum_c A^T \left[U_{n,t,c} \odot V_{c,f} \right] A. \quad (13)$$

This allows one to perform the summation over channels first (16 additions per tile per channel) and apply inverse transformation after the summation,

$$Y_{n,t,f} = A^T \left[\sum_c U_{n,t,c} \odot V_{c,f} \right] A, \quad (14)$$

thereby making its contribution to the operation count negligible. This results in net savings in the number of operations by a factor of $(36 + 36 + 4)/(16 + 16) = 2.375$[1].

The expectation of speedup in the fast algorithm due to reduced number of operations hinges on the assumption that the precomputation of the forward and backward transformation of data takes little time. In reality, our experiments revealed that the straightforward implementation of the above algorithm does not provide high performance, and platform-specific optimization is required.

4. TRANSFORMATION TO GEMM

In the above reasoning, we were making a silent assumption that the arithmetic throughput is the limiting factor of performance, i.e., memory traffic is completely overlapped with computation. This assumption holds true only if the order of tile convolutions is tuned to effectively re-use data in the processor's caches. To avoid this complexity, as shown [7], we we can express the arithmetic operations in Equation (14) as matrix multiplication, which allows us to delegate the complexity

[1] We factored in the need to do 4 additions per channel in the direct method

of overlapping memory traffic and computation to the GEMM function of a BLAS library.

Expressing the calculations through GEMM is possible because a transformed input tile $U_{n,t,c}$ can be reused to multiply with multiple corresponding filter tiles, and, similarly, a transformed filter tile $V_{c,f}$ can be reused to multiply with corresponding input tiles across all the batches. In addition, the input and filter tiles that are multiplied across C channels are accumulated into one output tile. Denoting the elements of the 4×4 matrix $U_{n,t,c}$ as $U^{x,y}_{n,t,c}$, and denoting elements of $V_{c,f}$ as $V^{x,y}_{c,f}$, we can define a 4×4 matrix $P_{n,t,f}$ with elements

$$P^{x,y}_{n,t,f} = \sum_c U^{x,y}_{n,t,c} V^{x,y}_{c,f}. \tag{15}$$

For each pair (x, y), Equation (15) expresses the multiplication of matrix $U^{x,y}$ by matrix $V^{x,y}$. The final result of convolution can then be written as

$$Y_{n,t,f} = A^T P_{n,t,f} A. \tag{16}$$

To further improve performance, we collapse multiple matrices $U^{x,y}$, which results in the first matrix in GEMM having greater number of rows.

5. Implementation in Code

To compute convolution with the fast algorithm we follow approach similar to that shown in [7]. There are three stages in this algorithm.

1. Input transformation: scattering on the image and filter data sets to form the input matrices.

2. Computation of product between transformed data and filter, and summation over channels expressed as GEMM, and

3. Output transformation: gathering the elements from the product matrices and their transformation to form the actual output of the convolution

Data transformation and the procedure for expressing the computation with GEMM are explained thoroughly in [7]. We have modified the data layout and way the input matrices for GEMM are formed and the pseudo codes in Algorithm 1 below illustrate the procedure.

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

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

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

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

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

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

The input data format is flexible and can be tuned with the help of merge factor M to achieve high GEMM performance. Here, $M = 1$ results in $NCHW$ format,

$M = N$ results in $CNHW$ format, and $(1 < M < N)$ shuffles N and C, keeping HW as fixed inner dimensions.

6. THE FALCON LIBRARY

Our implementation of image convolution with a 3×3 filter, codenamed FALCON (FAst paralleL CONvolution) is available under the MIT license on GitHub[2] The code contains initialization and cleanup routines and a single interface function for performing a convolution. The syntax of the routines is described in the header file included in the GitHub repository.

The code of the initial implementation is optimized to perform with high efficiency on Intel Xeon Phi x200 product family (formerly Knights Landing). Optimization measures that ensure high efficiency in the transformation step include:

1. Organizing the data structures in a way that, allows unit-stride access to input data and constant-stride access to output data;
2. Tiling the loops to maximize register data re-use in cores;
3. Unrolling inner loops to maximize the utilization of the register file and eliminate the dependence on the compiler estimate of the unroll factor;
4. Automatically vectorizing inner loops with compiler hints;
5. Tuning the count and affinity of OpenMP threads for maximum memory bandwidth;
6. Placing scratch data in the high-bandwidth on-package memory of the Intel Xeon Phi processor;
7. Tuning the inner dimension of data structures to be a multiple of 64 bytes (for aligned vector loads and stores), but not a multiple of 4096 bytes (to avoid cache associativity conflicts);
8. Pre-allocating and re-using scratch data structures to avoid dynamic memory allocation in computation.

[2] github.com/ColfaxResearch/FALCON

The code of the matrix multiplication step is optimized by:

1. Using the BLAS implementation of single precision matrix-matrix multiplication (SGEMM), which in our tests was linked to the Intel Math Kernel library (MKL);
2. Falling back to custom C language code instead of BLAS for tall and narrow (in column-major format) matrices, which are not handled efficiently by MKL;
3. Fusing multiplication of multiple matrices with low row counts into a single larger GEMM;
4. Using nested parallelism to process multiple matrix multiplications in parallel, with several threads working on each multiplication;
5. Tuning thread affinity and using the "hot teams" functionality in Intel OpenMP to persist the affinity within inner thread teams across parallel regions.

We ensured the functionality of the code and tuned performance only for the hardware and software configuration described in the next section. Special attention was paid to optimize the performance of the code for convolution sizes used in VGG Net [8].

7. PERFORMANCE

Results reported here are obtained on a 68-core Intel Xeon Phi processor 7250 with 96 GiB of DDR4 RAM and 16 GiB of MCDRAM in flat mode. The system is running CentOS 7.2 with stock kernel. The code was compiled with Intel C compiler 17.0.0.098 (Build 20160721) and linked with Intel MKL 2017 (build date 20160802).

To benchmark the convolution routine, we opted to construct a benchmark based on a practical application: the forward pass of VGG Net. For that purpose, we built a driver application that performs and times convolutions for input sizes corresponding to the 13 layers of the VGG Net configuration D [8]. We compare performance of FALCON with that of the convolution

operation of the DNN primitives module of Intel MKL. We tested the performance with a batch size of $N = 64$.

Our results are detailed in Table 1 and graphically presented in Figure 1. The x-axis in the plot is the time elapsed from the beginning of the calculation for a batch of N input images. The y-axis is the effective performance of each layer in Intel MKL (blue rectanges with dashed outline) and FALCON (yellow rectangles with solid outline). Rectangles corresponding to different layers are labeled in their corners with numbers from 1 to 13. Effective performance is measured in TFLOP/s. It is computed as the ratio of the number of operations in direct convolution, estimated as $2 \times (H-2)(W-2)RSNCF$, to the measured wall clock time, τ_{tot}. Labels indicate the total time of processing of all layers (0.42 s for FALCON, 0.47 s for MKL) and the corresponding effective performance (4.7 TFLOP/s for FALCON and 4.2 TFLOP/s for MKL).

Details of Figure 1 show that the direct method used in Intel MKL performs better than FALCON for the first 3 layers. Indeed, the tall and skinny matrix used in these first three layers results in poor performance of GEMM in FALCON. However, starting from layer 4, the method based on Winograd's algorithm used in FALCON is faster, and this performance advantage compensates for the time lost in the first three layers.

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

Table 1: Convolution performance in MKL and FALCON.

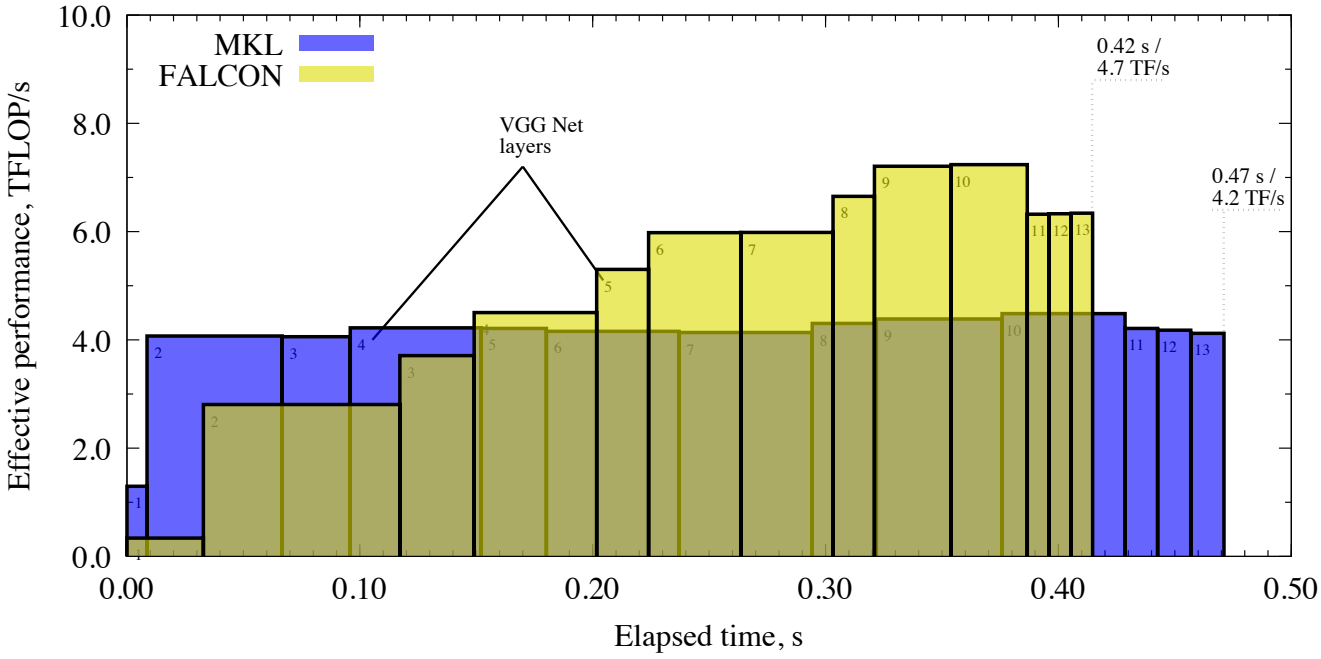


Figure 1: Convolution of VGG Net layers in FALCON and Intel MKL on Intel Xeon Phi 7250 processor (flat MCDRAM, quadrant mode).

Fast Convolution					FALCON in MCDRAM						FALCON in DDR4					
Layer	m	n	k	M	τ_{in}, ms	P_{in}, GB/s	τ_{out}, ms	P_{out}, GB/s	τ_{MM}, ms	P_{MM}, TFLOP/s	τ_{in}, ms	P_{in}, GB/s	τ_{out}, ms	P_{out}, GB/s	τ_{MM}, ms	P_{MM}, TFLOP/s
1	12544	64	3	1	1.3	149	14.1	292	17.4	0.28	4.1	47	60.2	68	85.4	0.06
2	12544	64	64	1	25.2	164	13.8	298	45.5	2.31	89.9	46	59.8	69	124.3	0.85
3	12544	128	64	4	6.3	166	6.9	296	18.7	2.81	22.4	46	30.3	68	51.4	1.02
4	12544	128	128	4	12.4	167	6.9	299	33.3	3.16	45.1	46	30.3	68	66.3	1.59
5	6272	256	128	8	3.2	163	3.5	290	15.6	3.38	11.4	46	15.0	69	27.3	1.93
6	6272	256	256	8	6.4	164	3.6	287	29.6	3.55	22.7	46	14.9	69	35.1	3.00
7	6272	256	256	8	6.3	166	3.5	290	29.7	3.54	22.8	46	14.9	69	35.1	3.00
8	3136	512	256	16	1.8	153	1.8	292	14.2	3.70	5.9	47	9.2	56	15.3	3.43
9	3136	512	512	16	3.7	149	1.7	296	27.4	3.84	11.8	47	7.5	69	29.9	3.52
10	3136	512	512	16	3.5	157	1.7	294	27.4	3.83	11.9	47	7.5	68	30.0	3.51
11	784	512	512	16	1.2	133	0.5	262	7.6	3.44	3.3	49	1.9	68	9.0	2.93
12	784	512	512	16	1.2	134	0.5	258	7.6	3.44	3.3	49	1.9	68	9.0	2.93
13	784	512	512	16	1.2	133	0.5	258	7.6	3.46	3.3	50	1.9	68	9.0	2.93

Table 2: Effective bandwidth of input and output data transformation and performance of matrix multiplication.

Based on the timing of MKL and FALCON performance, we argue that for specific DNN architectures, it may be possible to construct a hybrid convolution routine, in which for each layer either direct, or fast convolution is used, whichever is faster. In our example, using the direct method for the first 3 layers can save around 0.05 seconds, promising a total speedup over the direct method of $0.47/(0.42 - 0.05) \approx 1.3$.

Table 2 presents additional timing details. For each layer we report the time and effective bandwidth of input transformation τ_{in} and output transformation τ_{out}. We also indicate the size $\{m, n, k\}$ and the performance of the GEMM used for filter application. Two sets of results are reported: with FALCON and the benchmark pinned to the high-bandwidth on-package memory (MCDRAM) and the on-platform memory (DDR4).

For high-bandwidth memory benchmarks, because our data structures were less than 16 GiB in size, we could fit them in the available MCDRAM. The processor was in flat memory mode, and so we ran the entire application in MCDRAM by setting the default NUMA policy with the `numactl` tool [9]. We also tested performance with the processor in the cache memory mode, and observed similar results. However, running the calculation in the on-platform memory (DDR4), we observed performance degradation by a factor of 2.5.

With MCDRAM, the input transformation achieves between 130 and 160 GB/s of memory access bandwidth. This is only around 30% of the bidirectional MCDRAM bandwidth. This is related to asymmetric traffic (more writes than reads), scattered memory pattern, and the presence of computation mixed in with data access. The output transformation achieves a better performance between 260 and 300 GB/s. In our experiments, the data layout that we ended up using optimizes the memory bandwidth as well as the overall timing.

The information about GEMM performance shows that with MCDRAM, it achieves between 3.2 and 3.8 TFLOP/s for layers 4-13, which is a large fraction of SGEMM performance in the ideal case of large square matrices (we measured 4.5 TFLOP/s). However, for layers 2 and 3, GEMM achieves only 2.3-2.8 TFLOP/s due to the small size of the inner matrix dimension, k. For layer 1, $k = 3$, and this computation is memory-bound. This is the case where we used custom C code instead of the BLAS call because in this case MKL delivered significantly worse GEMM performance.

Future optimization should focus the input transformation, as it operates at a low efficiency compared to its theoretical peak value. At the same time, the output transformation and GEMM are performing well. Poor performance in the first 3 layers may be ignored as MKL can be used in place of FALCON in this case.

Timing information in Table 2 shows that the memory-bound data transformation takes around 30% of execution time with the compute-bound GEMM taking the rest. We speculate that additional performance improvement may be obtained by splitting the batch of images into several sub-batches and overlapping in time the data transformation and GEMM computation.

8. CONCLUSION

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

8.1. PERFORMANCE OPTIMIZATION

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

8.2. HIGH-LEVEL LANGUAGE

Despite the complexity of code optimization, the FALCON code does not use any assembly or intrinsic functions for explicit access to platform-specific instructions. Instead, it relies on automatic vectorization in the compiler, on standard functionality of the OpenMP framework, and on traditional BLAS routines. This simplifies future code maintenance and adaptation of the application to the upcoming computing platforms. Additionally, our case study proves by example the possibility of using high-level languages and frameworks in computational applications for Intel Xeon Phi processors.

8.3. SPEEDUP OVER DIRECT METHOD

In the context of machine learning, we achieved convolution performance greater than that of the direct method implemented in the industry-leading mathematical library for Intel Xeon Phi processors. The performance advantage of approximately 10% was measured for a workload simulating VGG Net forward pass. Based on our argument for hybrid approach combining direct and fast algorithms (see Section 7), the speedup for this ConvNet may be improved to 30%. In some layers of VGG Net, FALCON is faster than MKL by as much as 50%, so the application of Winograd's algorithm to convolution in other DNN architectures may yield even more significant speedups.

8.4. IMPORTANCE OF HIGH-BANDWIDTH MEMORY

According to our comparison testing, high-bandwidth memory is the key element of the Intel Xeon Phi processor architecture that makes fast convolution perform better than the direct method. This is not an obvious result because ML tasks are generally considered compute-bound. However, as long as upcoming models of Intel Xeon Phi products retain the MCDRAM, they can benefit from fast convolution. In particular, performance advantage of fast convolution may develop strongly in the upcoming Knights Mill architecture specifically tuned for deep learning applications [10]. In addition, the upcoming coprocessor form-factor of Intel Xeon Phi coprocessors is a suitable platform for ConvNets with fast convolution. Indeed, the data structures used for our VGG Net benchmark are under 16 GiB in size. This is suitable for offloading calculations to coprocessors, assuming that they are manufactured with at least the same amount of MCDRAM as their bootable counterparts.

8.5. APPLICATION TO MACHINE LEARNING

To our knowledge, the FALCON library is the first open-source implementation of fast convolution for Intel Xeon Phi processors. We publish it under a permissive MIT license[3] in hopes that the high-performance computing community can contribute to the improvement of the code and to its adoption in production machine learning libraries.

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

[3] github.com/ColfaxResearch/FALCON

ACKNOWLEDGEMENTS

We thank Alexander Heinecke (Intel) for his review
that led to a bug fix in reported performance values.

REFERENCES

[1] Nicole Hemsoth. The next wave of deep learning applications.
http://www.nextplatform.com/2016/09/14/next-wave-deep-learning-applications/.

[2] Wikipedia. Convolutional neural network.
https://en.wikipedia.org/wiki/Convolutional_neural_network.

[3] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2016.
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.137.482.

[4] Intel Math Kernel Library.
https://software.intel.com/intel-mkl.

[5] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. 1997.
http://www.dspguide.com/ch24/2.htm.

[6] Shmuel Winograd. Arithmetic complexity of computations.
http://dx.doi.org/10.1137/1.9781611970364.

[7] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks, 2015.
https://arxiv.org/abs/1509.09308.

[8] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2014.
https://arxiv.org/abs/1409.1556.

[9] Colfax Research. Hands-On Workshop "Performance Optimization for Intel Xeon Phi x200 Product Family" (HOW Series), 2016.
http://colfaxresearch.com/how-knl/.

[10] top500. Intel Unveils Plans for Knights Mill, a Xeon Phi for Deep Learning, 2016.
https://www.top500.org/news/intel-unveils-plans-for-deep-learning-processor-in-xeon-phi-lineup/.