– Independent Work Report Fall, 2015 –

Performance Optimization for CNNs on Modern Intel CPUs

Vladimir Feinberg
Adviser: Kai Li

January 5, 2015

Abstract

Speeding up training and prediction for Convolutional Neural Networks (CNNs) benefits the computer vision field because of the networks' recent widespread adoption. The introduction and usage of deep networks has also been growing popular, but comes at the cost of longer processing time. A CPU-based CNN package, ZNN, is analyzed to identify and address bottlenecks in the context of Modern Intel CPUs, including the Xeon Phi Knight's Corner, Xeon E5, and Core i7. For sufficiently wide networks, the computation of the Discrete Fourier Transform – used for linearithmic-time convolutions – dominates runtime. By considering memory utilization, cache usage, and vectorization intensity behaviour, this work introduces several modifications. The main contributions are the vectorization of prime-sized Fast Fourier Transforms and a novel algorithm that computes a rotation of the Fourier Transform that is optimized for the single-thread larger-than-cache 3D use case. These changes resulted in up to 14.3% improvement across the various platforms considered for neural nets of varying topologies.

Introduction

Motivation and Goal

CNNs are instances of a supervised machine learning algorithm for image classification. This paper focuses on the training of CNNs for 3D image classification, which has recently been applied to improve on classification accuracy for traditionally anisotropic 2D image classification situations because of the additional context that the 3D information provides [24]. Notably, such techniques are used in video classification [19] and image segmentation, particularly for fMRI imaging of the brain, aiding in connectome reconstruction [7].

As such, there is much to gain from performant CNNs because they aid the entire computer vision field in general.

Implementations of CNNs all follow the approach first proposed by [28] – gradient descent for a loss function, which is a differentiable expression for classification accuracy. The CNN can be thought of as the composition of many various differentiable functions, and backpropogation is the repeated application of the chain rule; in essence, applying Newton's method to the parameters of each unit within the CNN (a neuron) to find a local minimum for loss.

For the data scientist preparing a CNN model, speed in both evaluating the CNN prediction (the forward pass) and updating CNN parameters based on loss (the backward pass) is crucial for exploring more images and different models to improve accuracy.

This paper focuses on ZNN, a CPU-based CNN implementation [36]. Various modern Intel architectures were considered, including the Xeon E5, Xeon Phi Knight's Landing, and Core i7.

Roadmap

The format of this paper is as follows. Section 2 provides an overview of the topics at hand. Next, section 3 describes prior related work. Then, sections 4 and 5 detail the approach and exploratory analysis taken to decide what changes to make. Sections 4, 6, and 7 describe the implementation and evaluation of improvements offered. Finally, section 8 summarizes the work and explores future potential, and section 9 provides acknowledgements.

Background

CNNs

CNNs are composed of a series of layers [36]. Each convolution layer contains a set of input and output feature maps, which are each 3D images in greyscale. The initial input feature map is just the original image itself. Output feature maps are sums of convolutions of input feature

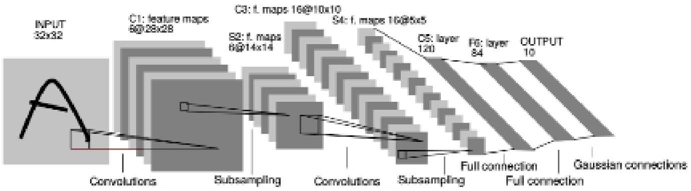


Figure 1: LeNet-5, the architecture for the CNN for OCR in [23], featuring a variety of layer types. Subsampling and max-pooling are analogous.

maps. Neural networks are comprised of other operations as well – including linear and nonlinear rectifiers, max-pooling filters, which sparsify the image by extracting only maximal elements of a neighborhood of voxels, and fully-connected layers – as shown in figure 1. In the class of networks examined, which are moderately-sized neural networks on a single machine, the convolution layers dominate performance; this is in contrast to distributed large networks, where fully-connected layers' communication occupies most of the time [20]. We let the *depth* of a network refer to the number of layers it has.

As described in [26], for a certain convolution layer with f input features and f' output features, our forward pass computes our outputs $y_{f'}$ given weights $w_{ff'}$ and inputs x_f:

$$y_{f'} = \sum_f x_f * w_{f'f} \tag{1}$$

After going through the network, the network returns with a backward pass to find the per-input blame for a loss function L:

$$\frac{\partial L}{\partial x_f} = \sum_{f'} \frac{\partial L}{\partial y_{f'}} * w_{f'f}^T \tag{2}$$

Finally, we update the weights by finding their loss gradient:

$$\frac{\partial L}{\partial w_{f'f}} = \frac{\partial L}{\partial y_{f'}} * x_f \tag{3}$$

For additional clarity, here are the dimensions (assume a kernel size $k \times k \times k$ and inputs $n \times n \times n$ with $n' = n - k + 1$): $x_f \in \mathbb{R}^{n \times n \times n}, y_f \in \mathbb{R}^{n' \times n' \times n'}, w_{f'f} \in \mathbb{R}^{k \times k \times k}$. $\partial L / \partial u$ has dimensions of u as L is real-valued.

Typical CNN sizes were informed by the winning networks for the competition ISBI [7], which provided the real-life data for fMRI image segmentation as well. The winning network in 2012 had only 4 convolutional layers (as well as some filters and fully connected ones), but as the network sizes have increased in the past the typical network focussed on was 30 feature maps wide and 10 layers deep [10]. Others had typical depths [21].

The Fourier Transform

The convolution operation $(a * b)_{ijk} = \sum_{lmn \in \mathrm{dims}(b)} a_{(i-l)(j-m)(k-n)} b_{lmn}$ (indices modulo $\mathrm{dims}(a)$) operates in $O(N^3 M^3)$ time for cubes a, b of lengths N, M, respectively. For large kernels, we rely on the Convolution Theorem (4) to cut this down to $O(N^3 \log N)$ time, assuming $N \geq M$ [26]:

$$a * b = \mathrm{3DIDFT}(\mathrm{3DDFT}(a) \times \mathrm{3DDFT}(b)) \tag{4}$$

Above, $\mathrm{3DDFT}(\cdot) \in \mathbb{C}^{N \times N \times N}$ is the Three Dimensional Discrete Fourier Transform and 3DIDFT is its inverse, with a pointwise multiply in the frequency domain saving computation for large kernels. A multidimensional DFT is equivalent to a single-dimensional DFT along each axis: see equations (5) and (6) for comparison [22].

$$\mathrm{1DDFT}(x)_j = \sum_{k=0}^{N-1} x_k \exp\left(\frac{-2\pi i j k}{N}\right) \tag{5}$$

We let x_i be the i-the element of a one dimensional signal of length N in the 1DDFT equation, and let x be a three-dimensional one in (6). In the following, we let $\omega_i = \exp(-2\pi i/N)$.

$$\mathrm{3DDFT}(x)_{jkl} = \sum_{m=0}^{N-1} \omega_j^m \sum_{n=0}^{N-1} \omega_k^n \sum_{o=0}^{N-1} \omega_l^o x_{mno} \tag{6}$$

Both (5) and (6) display the naive version of the DFT algorithm, which would take quadratic time to compute, but they capture the difficulty of high dimensional DFT: the data access for computing a single element requires traversal across all axes.

To compute the DFT and IDFT, we rely on the Fast Fourier Transform, which reduces the FFT into subproblems by decomposing a composite size-n_1n_2 DFT into n_1 size-n_2 problems (or vice-versa). Small primes with hard-coded DFTs are the base cases, and large primes can be converted into a composite-sized problem of similar size via a cyclic convolution [17].

We notice that we may cache the FFTs to save recomputation. ZNN implements this caching [36]. If we compute all f x_f, ff' $w_{ff'}$, f' $y_{f'}$, and f' $\partial_{y_{f'}}L$ FFTs once, then we can have significant re-use. The forward pass (1) over uses each $w_{ff'}$ once and each x_f f' times. The backwards pass (2) uses each $w_{f'f}$ once and each $\partial_{y_{f'}}L$ f times. Finally, the weight update (3) uses each $\partial_{y_{f'}}L$ f times and x_f f' times.

Note that we may further save on additional transforms by relying on linearity of the Fourier transform operator. Denoting $A_i = 3DDFT(a_i), B_i = 3DDFT(b_i)$

$$\sum_i a_i * b_i = \sum_i 3DIDFT(A_i \times B_i) = 3DIDFT\left(\sum_i A_i \times B_i\right) \tag{7}$$

Thus we may save some time on inverse transforms. ZNN implements the optimization in (7) for both (1) and (2). Note that this implies that the operations performed in the frequency domains are only pointwise additions and multiplications.

Intel Processors

Recent Intel CPUs offer the potential for high vectorization, or multiple simultaneous floating point operations with SIMD instruction extensions, but these require carefully addressing the memory access requirements in a constrained-memory environment: a program must be computationally intensive enough to make full use of the processing power that the CPUs provide [1] [5] [4].

The Xeon Phi thus has the most raw computational power, but is most memory constrained. Note that while the Xeon Phi has a large per-core L2 cache, it is shared between more hyperthreads. Furthermore, the Phi has no L3 cache. Finally, empirical observations found that satisfying a local on-core miss from a remote L2 cache is almost as slow as going to memory the former is 250 cycles and the latter 302 [15]. For this reason, engineering correct cache-conscious algorithms that utilize vectorization will be important.

Family	Xeon Phi (Knight's Landing) [6]	Xeon E5 [3]	Intel Core i7 [2]
Cores x Hyperthreads	61x4	32x2	4x2
Clock speed	1.1GHz	2.6GHz	2.6GHz
L1D+I	64KB	64KB	64KB
L2	512KB + remote	256KB	256KB
L3	0	20MB	4MB
Vectorization	16x4B, mMIC	4x4B, mAVX	8x4B, mAVX2

Figure 2: Overview of tested Intel architectures

Related Work

CPU vs. GPU based networks

The focus on a CPU rather than GPU implementation for CNNs in this work is based off the following propositions for the future of CNN technology: CPUs will continue to grow in cores as a means of increasing processing power, and while GPU parallelism already fully saturates the scalability of moderately-sized networks, CPUs still have more parallelization potential and their implementation with ZNN demonstrates near-linear scalability up to physical core count [36]. In addition, CPU-based implementations allow higher levels of abstraction for task completion and memory access compared to the GPUs SIMD threading model – in turn, we are able to implement more flexible scheduling policies and computation-saving max-pooling techniques [36].

The vast majority of current CNN implementations rely on direct convolution and GPUs, which are particularly well suited to SIMD tasks. The CUDA extension to the popular Torch [33] package, `cunn` [32], is representative of the current package environment [9]. These approaches rely on the high SIMD scalability of direct kernel convolutions. In particular, packages like [32] are vulnerable to the algorithmic disadvantage displayed above for large kernel sizes k. [36] demonstrates that ZNN outperforms its GPU counterpart for kernels of size 7x7x7. We see that we have more efficiency in the CPU implementation in figure 3. The figure also demonstrates the underperformant Xeon Phi on multiple network topologies compared to the Xeon.

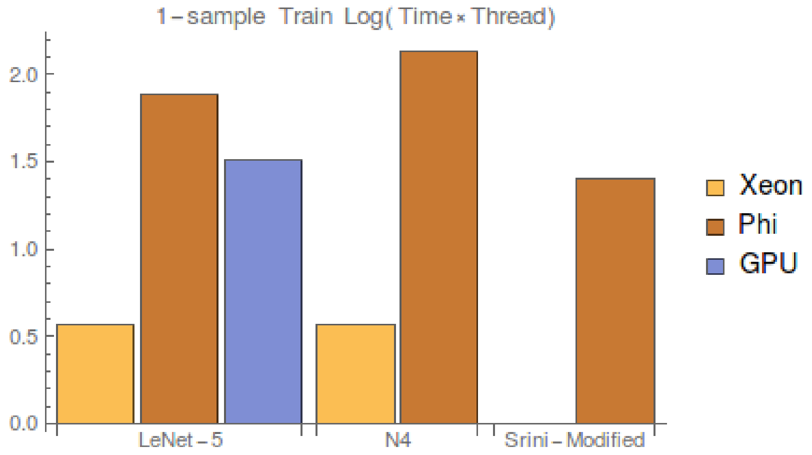


Figure 3: Log-graph of total thread-time. Topologies run on are LeNet-5, an adapted version of the 2D 7-layer deep net from [23]; N4, the winning 10-layer 2D net from the ISBI competition [11]; and Srini-Modified, a custom, artificial 6-layer 3D convolution-only network. LeNet-5 was run on the MNIST OCR data set and N4 and Srini-Modified were run on the ISBI data set (which provides 3D context). Each architecture was run with the maximal number of threads. The GPU test was on a Nvidia GTX Titan card, which enables 15 multiprocessors at 2048 threads each. The time above is average time to train 1 sample (stochastic gradient descent, forward, backward, and update passes). In Xeon and Xeon Phi, 25000 iterations were run. For GPUs, one trial of 60000 iterations. The varying sample sizes were due to different resource amounts.

FFTs

In this analysis, FFTW 3.3.4 (on the Xeon and Xeon Phi) and 3.3.3 (on the i7) are used for computing FFTs. Unfortunately, Intel's MKL, a native vendor-provided library, was not an option because it is closed-source, so it could not be manually optimized.

FFTW has been developed over two decades and has come to implement a variety of strategies in computing FFTs. The Cooley-Tukey algorithm from [12] can be implemented with different radices (factors for decomposition), with strategies for decimation in the time as well as frequency domains [17]. Furthermore, there are many cache-conscious improvements such as buffering intermediate strided input (e.g., when taking an FFT along a non-contiguous dimension) [18]. Code generation is used to create codelets for small DFTs with appropriate prefetching and vectorization [17].

The subject of creating faster FFTs has been widely studied as well. 3D FFTs may be decomposed into 4-dimensional problems that are parallelizable with only a few rounds of communication.

On the Xeon Phi, [25] develops a two-pass approach dividing along the exterior dimension for blocking. For medium-to-large 3D FFTs (256x256x256), this algorithm can utilize the entire Xeon Phi well, outperforming the native vendor MKL library by a factor of 2.2. However, the key ingenuity in [25] leverages all processors in the MIC architecture. In the CNN, for a layer with f input and f' output feature maps, there are ff' possible concurrent FFTs to do [36] (let the average value f amongst all layers be the *width* of the network). These fully saturate available parallelism already, so there is no need to stray from a single-threaded FFT algorithm.

For large 1D FFTs, a variety of cache-optimal [18] and vectorized cache-blocking [31] [8] algorithms have been designed as well. Unfortunately, with kernel sizes tending to stay under 10 on each side and network depths below 30 layers, the individual FFT along a single dimension does not grow to more than several hundred points, so the data for a single FFT already comfortably fits into L1.

The use case examined, with 3D FFTs of size ranging from several KB to several MB on a single thread with the cache sizes described in the previous section, does not have much exploration. However, some techniques, such as using multiple passes on data with intermediate rotations, still apply.

Approach

General Methodology

In the Full Correlation Matrix Analysis toolkit, specialization for the Xeon Phi has earned remarkable results, with 5 to 16 fold speedup compared to a baseline generic CPU implementation relying on Intel's MKL [34]. This speedup was achieved by changing the matrix blocking algorithm from Intel's MKL policy to a custom one, smarter cache retention policies, and remodelling data structures for better vectorization.

Influenced by the above and additional discussions with my advisor, my approach will be guided by looking for potential improvements in the following areas:

a. Fully utilize all available cores and HW threads

b. Increasing L2 cache hits - blocking, data rearrangement

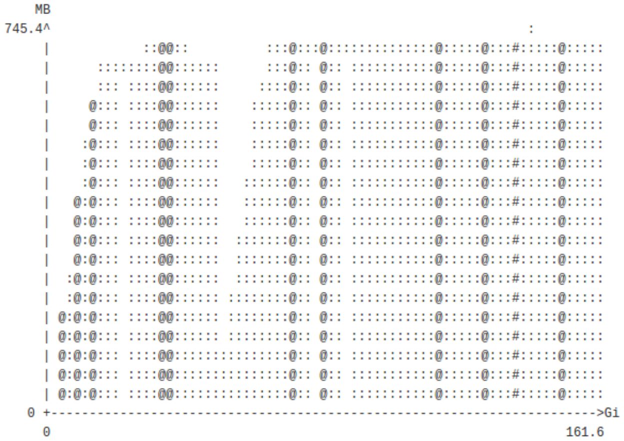
c. Increasing Vectorization Intensity - contiguous data, vectorizable algorithms

d. Intelligent RAM use - structural changes to data organization and allocator.

Exploration

Memory Exploration

Using `valgrind`'s `massif` tool, one can construct a memory profile for the program at various time points 4 [30].



```
    MB
745.4^                                                              :
     |            ::@@::          :::@:::@::::::::::::::@:::::@:::#:::::@:::::
     |      ::::::::@@::::::      :::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |      ::: ::::@@::::::     ::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |     @::: ::::@@::::::    :::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |     @::: ::::@@::::::    :::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |    :@::: ::::@@::::::    :::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |    :@::: ::::@@::::::    :::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |    :@::: ::::@@::::::   ::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |   @:@::: ::::@@::::::   ::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |   @:@::: ::::@@::::::   ::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |   @:@::: ::::@@::::::  :::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |   @:@::: ::::@@::::::  :::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |  :@:@::: ::::@@::::::  :::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     |  :@:@::: ::::@@:::::: ::::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     | @:@:@::: ::::@@:::::: ::::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     | @:@:@::: ::::@@:::::: ::::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     | @:@:@::: ::::@@:::::::::::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     | @:@:@::: ::::@@:::::::::::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
     | @:@:@::: ::::@@:::::::::::::::@:: @:: :::::::::::@:::::@:::#:::::@:::::
   0 +----------------------------------------------------------------------->Gi
     0                                                                   161.6
```

Figure 4: Memory consumption over time (measured in gigainstructions) for several training iterations of a 30-width 10-depth CNN. In this case, the network itself takes less than 10% of the memory. Most of the 0.75GB used here come from kernel matrices.

Analyzing `massif` results and counting FFT reuses, one finds that for a network of width W, the inputs x_f and gradients $\partial_{y'_f} L$ from equations (1), (2), and (3) are reused $O(W)$ times while the kernel FFTs are reused only once (computed in the forward pass, the retrieved in the backwards one).

However, there are $O(W^2)$ kernel FFTs, so they dominate FFT timing. Caching them, when feasible, leads to a 30% improvement [35]. This is because together they account for most of the FFTs performed, and the caching halves computation.

The Xeon Phi is limited in its RAM memory, so it is not able to cache FFTs for kernels for very large networks. Though these are not the focus of this work. Because the re-use of the kernel FFTs is deterministic and follows a linear pattern through the network (we compute the FFTs as we go up the network, then reuse them in reverse order exactly once), there is potential here for an optimal caching policy that offloads FFTs via DMA to a remote large RAM, but then returns them back to the local Phi memory – such a policy would have to consider the varying latencies between the DMA speed and propagation along the network.

VTune Exploration

Using Intel VTune, we examined the relevant hardware event counts for vectorization intensity and cache performance to optimize for the intel processors [4].

Benchmarks are run on W-width and D-depth networks, composed only of convolutional layers. The data used to train the networks is random – this does not affect computation bottlenecks because the operations performed on the data are the same: see figure 5.

Data	Random	ISBI [7]
Hotspot 1	40.1%	40.2%
Hotspot 2	31.3%	31.3%
Hotspot 3	19.6%	20.4%
Hotspot 4	5.8%	5.9%

Figure 5: VTune hotspots (by cycle count in procedures). The procedure names are obscure, but all four are from the FFT library. Procedures with less than 5% of computation time are not shown

VTune analysis was mainly performed on the Phi. Xeon analyses were used iteratively as a faster alternative when optimizing cache performance during the development of the transpose algorithm, but VTune did not support vector floating point unit hardware events on the E5. The initial analyses below were run on the Xeon Phi with 60 threads.

On a width-30 depth-10 network, VTune demonstrated a consistently low vectorization intensity, at 1.8 of 16 floating point registers filled on average throughout 75 training iterations. Reducing the depth of the CNN would serve to increase the cache hit ratio. This would reveal if the lack of vectorization is due to poor cache performance alone. Reducing to depth 5 increased intensity to only 2.3 of 16 units, which indicated that the code is not sufficiently vectorized. This analysis also revealed that FFT routines occupy at least 66% of the total runtime. As networks get wider and caching kernel FFTs is disabled, this figure increases.

The cache performance itself was also fairly poor. On the depth-5 network, only about 34% of the L1 misses were serviced by the local L2 cache. The remaining 66% were serviced by a remote L2 or memory, but both are undesirable (remote L2 contributed to 73% of that remaining proportion). We see this effect exacerbated by increasing thread count, which further reduces effective cache size as the number of hyperthreads per CPU (and therefore per L2 cache) increases 6.

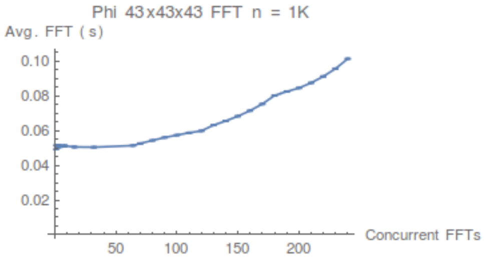


Figure 6: Note the increase in computation time for the FFT as the number of threads increases. This is due in part to higher operating system overhead in scheduling, but it is mostly caused by shared compute and cache resources.

One immediate attempt to improve upon the cache performance was to create a throttling scheduler; that is, to not launch tasks if there would be insufficient cache space. Recall that the three tasks, forward, backward, and update from (1), (2), and (3), have a predictable amount of memory usage. The throttling scheduler added a condition variable that checked for a sufficient amount of cache space to open before launching the next task. The supposition was that the improved cache performance would eclipse the relinquished compute resources. Unfortunately, this proved not to be the case, even though there was a 15% local L2 hit ratio improvement on a width-30 depth-10 128-thread CNN.

Implementation

Vectorization of Rader's Algorithm

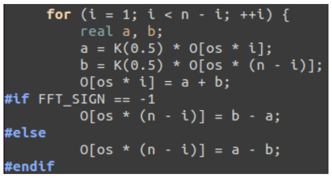
FFTW's generated codelets are already vectorized for small FFT sizes (thus, for the sizes of FFTs considered, below 256, most may be expressed as twiddled codelet FFTs on the factors defined by the Cooley-Tukey algorithm; for instance, an FFT of size 100 is decomposed into an outer-loop size-10 FFT and 10 inner-loop size-10 FFTs [12] [17]). However, for primes that don't exist as codelets, one must employ Rader's algorithm.

FFTW's default codelets are for sizes 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 25, 32, 64, and 128. Thus, starting from primes sizes 17 and up, Rader's algorithm is invoked.

Because the rexpression as a composite FFT problem takes linear time, the prime sizes are significantly slower than composite ones for FFTs. However, it is inconvenient to increase the FFT width to a composite number – because a neural network changes in size from large to small as one traverses up its layers, the arithmetic progression of sizes frequently generates some prime sizes.

FFTW does not have Rader's algorithm written in a vectorizable manner; in fact, generating a vectorization report for the `icc` (Intel C Compiler) revealed that many of the FFT strategies in FFTW are not vectorized. VTune exploration revealed a hotspot within the core of Rader's algorithm.

To vectorize the code, the algorithm needed to be re-expressed using the array-slice notation provided by Intel Cilk Plus compiler extension [27] [13]. Those familiar with functional programming may notice that this re-expression is a re-formulation into map and reduce steps. For a small example, see figures 7 and 8.

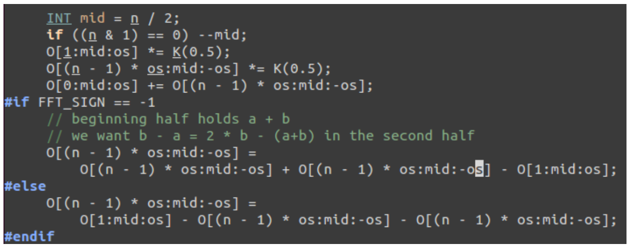


```
        for (i = 1; i < n - i; ++i) {
                real a, b;
                a = K(0.5) * O[os * i];
                b = K(0.5) * O[os * (n - i)];
                O[os * i] = a + b;
#if FFT_SIGN == -1
                O[os * (n - i)] = b - a;
#else
                O[os * (n - i)] = a - b;
#endif
```

Figure 7: An excerpt of code from FFTW before vectorization. Note the for loop is expressed in a manner that is not obviously linear to the compiler. Furthermore, the paired element accesses prevent accurate prefetching and any vectorization at all. The `K` macro ensures correct width of literals and `real` is a 4-byte float.

Transpose Algorithm

The next optimization performed is inspired by the transpositions used in 2D decompositions of distributed parallel 3D FFTs [29]. Developed with the goal of reducing L2 miss ratios, we make the following observations for $N \leq 256$ and ZNN:



```
    INT mid = n / 2;
    if ((n & 1) == 0) --mid;
    O[1:mid:os] *= K(0.5);
    O[(n - 1) * os:mid:-os] *= K(0.5);
    O[0:mid:os] += O[(n - 1) * os:mid:-os];
#if FFT_SIGN == -1
    // beginning half holds a + b
    // we want b - a = 2 * b - (a+b) in the second half
    O[(n - 1) * os:mid:-os] =
        O[(n - 1) * os:mid:-os] + O[(n - 1) * os:mid:-os] - O[1:mid:os];
#else
    O[(n - 1) * os:mid:-os] =
        O[1:mid:os] - O[(n - 1) * os:mid:-os] - O[(n - 1) * os:mid:-os];
#endif
```

Figure 8: The same excerpt, vectorized using Intel Cilk Plus array notation. Note one needs to change the operations performed at times. The notation $A[x:n:\delta]$ refers to a strided array slice over elements $x+i\delta$ for $i \in \{0,1,\ldots,n\}$.

a. On average, $2N^2$ fits comfortably within the L2 caches (see figure 2). This means there's enough room for the input and output for a 2D transform.

b. The only operation we apply to the frequency domain is the pointwise multiplication and addition from (1) and (2).

c. Pointwise operations are invariant under rotation.

d. The final FFT along the outer dimension of the cube is strided with a step of N^2

e. ZNN has a size-stratified lockfree slab allocator, which is designed to handle many concurrent requests of standard size well, compared to a lock-based standard allocator[36].

The format of the 3D image is as a flattened array. If we let the axes be labelled xyz, this implies that the element ijk is at position $iN^2 + jN + k$.

In the following, $a-b-c$ implies the allocation of a cube c of size abc. We fix the initial axes to be $x-y-z$ for the real input data. Thus, denote $c[i]$ as the i-th yz-plane elements, $c[i][j]$ as the i,j-th z-axis along the cube, and $c[i][j][k]$ as its k-th element.The real data type is a 4 byte float, and the complex one is two 4 byte floats. Further, denote $z^* = \lfloor z/2 \rfloor + 1$, which is the number of elements we need to store in the z dimension in the frequency dimension, relying on the property that $3DDFT\left(\mathbb{R}^{N\times N\times N}\right)$ is a Hermitian set, so we don't need to store redundant information.

Algorithm 1 describes the behaviour of invoking FFTW directly for the 3DDFT. 2DDFT(I,O) is another call into another vanilla FFTW plan for two dimensional arrays (input I and output O). 1DDFT is the one-dimensional analogue. This may or may not use an indirect buffer for the strided (y) dimension, depending on the measurements that FFTW performs when creating the first FFT

Algorithm 1 Original 3DDFT algorithm from FFTW

```
function 3DDFT(cube of reals $R\ x - y - z$)
    Allocate complex cube $C\ x - y - z^*$
    for each $i \in \{1, ..., x\}$ do
        2DDFT$(R[i], C[i])$
    end for
    Deallocate $R$
    Allocate a buffer $b$ of size $x$.
    for each $j \in \{1, ..., y\}$ do
        for each $k \in \{1, ..., z^*\}$ do
            Copy $b = C[*][j][k]$
            1DDFT$(b, C[*][j][k])$
        end for
    end for
    return $C$
end function
```

execution plan. Typically, for sizes too large for input and output to fit in the L2 cache, it will choose to do this.

The observations enumerated above suggest that it may be useful to first do a 2D transform along the inner dimensions (the first of which is contiguous). Furthermore, it is important to introduce intra-dimensional padding. In other words, one ought to expand the innermost dimension z to size $p(N)$, where the padding operation for an alignment A is $p(x) = \lceil N/A \rceil \cdot A$. Xeon Phi uses an alignment of 64 bytes and the other architectures use 16 – such alignment is essential to fully populate the vector floating point unit, which rejects maligned input, forcing sequential floating point evaluation. See algorithm 2 for an implementation.

Note that the transpose algorithm does not ever significantly increase the total amount of allocated space – algorithm 2 uses at most $M_2 = 2xyp(z^*) + 2p(x)yz^*$ floats and algorithm 1 uses $M_1 = xyz + 2xyz^*$. With $x = y = z = N$, $M_2/M_1 = o(1)$.

Algorithm 2 The transpose algorithm invokes the same 2DDFT routines.

```
function ROTATED-3DDFT(cube of reals $R\ x-y-z$)
    Allocate complex cube $C\ x-y-p(z^*)$
    for each $i \in \{1,...,x\}$ do
        2DDFT$(R[i], C[i])$
    end for
    Deallocate $R$
    Allocate a complex cube $T\ y-z^*-p(x)$.
    for each $j \in \{1,...,y\}$ do
        for each $k \in \{1,...,z^*\}$ do
            1DDFT$(C[:][j][k], T[j][k])$
        end for
    end for
    Deallocate $C$
    return $T$
end function
```

Evaluation

Vectorization Improvement

Vectorizing Rader's algorithm provides an improvement in both vectorization intensity and overall time for FFT computation on sizes that invoke Rader's algorithm, on large primes (≥ 17) and on values that call on procedures for such primes in their recursions (small multiples of such primes, since $N \leq 256$).

A VTune analysis on the Phi demonstrated that for a size-43 cube, FFTs after vectorization had an increased vectorization intensity (from 1.257 of 16 registers filled to 6.174 averaged over 1000 runs), as expected (some unvectorized operations remain, since only the time-dominating functions were vectorized, and reduce operations have some inherent sequential dependence. However, there were no speed gains (16). The same analysis explained that this was likely caused by a lowering of L1 hit ratio (0.994 to 0.988) as computational intensity increased. This was exacerbated by a large L1 servicing latency (over 700 cycles for both programs), indicating a need for improved manual prefetching. Averaged across all prime-sized FFTs of size less than 256, performed 10 time each, vectorization alone actually slows down the FFTs by 2.7%.

Vectorization benefits hold with scale on the Xeons – see figure 10.

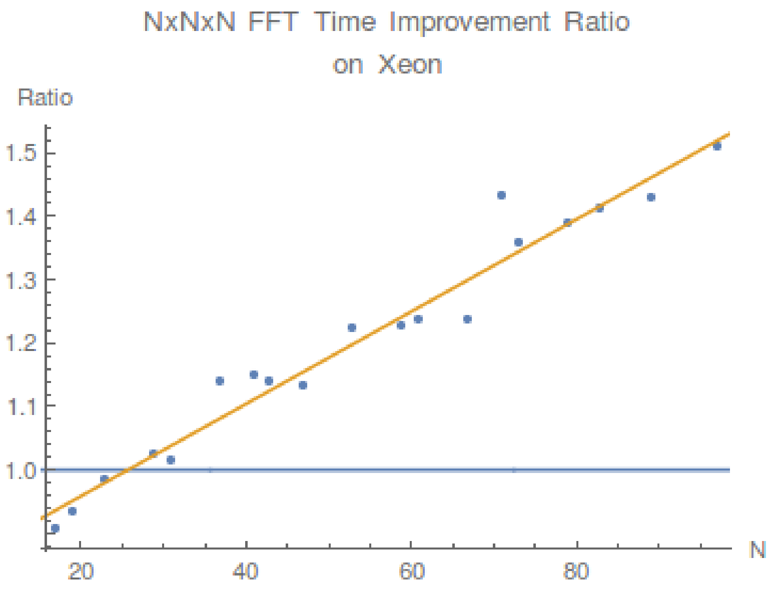


Figure 9: Time improvement ratio before and after introducing vectorization on the Xeon. An forward-transform FFT was computed 50 times, and the average time was taken. The fitted line's slope is 0.00728 and its intercept is 0.812. The line's $R^2 = 0.958$.

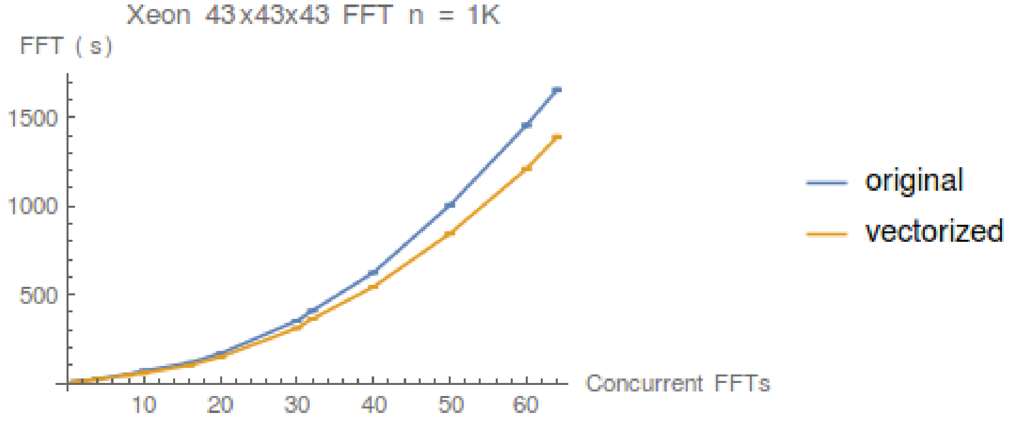


Figure 10: 1000-run average FFT time, forward only, on the Xeon.

Transpose Algorithm Improvement

On the i7 (figure 11) and Xeon (figure 12), we see modest performance improvements for the raw FFT computation. The below figures use the time improvement ratio (ratio of old time to new time) to evaluate performance gains across a variety of FFT sizes.

On the Xeon, we observe the theorized improvement in L2 cache hits, displayed in figure 13.

The transpose algorithm alone fares fairly well in the context of the neural network. On wider

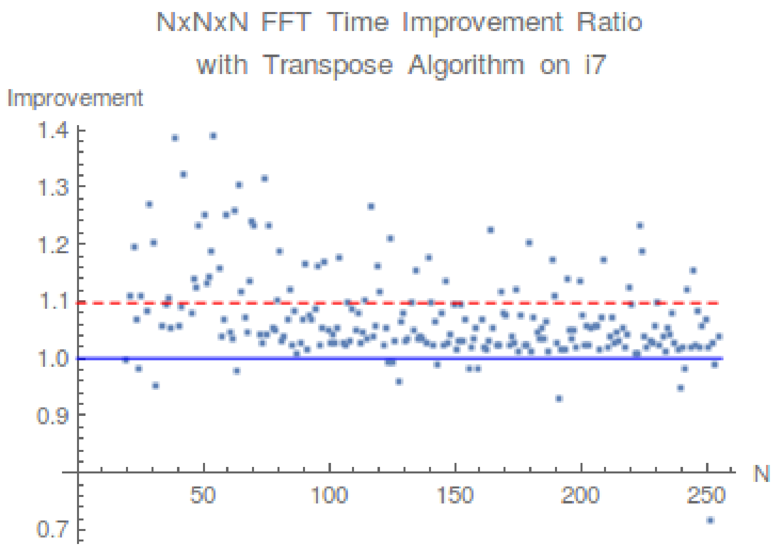


Figure 11: Time improvement ratio before and after introducing the transpose algorithm on the i7. An FFT was computed 10 times, and the average time was taken. The average duration before for a given size was divided by the average duration with the transpose algorithm. The mean improvment across all sizes is 9.7%. This is only for the forward FFT operation. The solid blue line is unity, indicating no effect on performance. The dashed red line is the mean

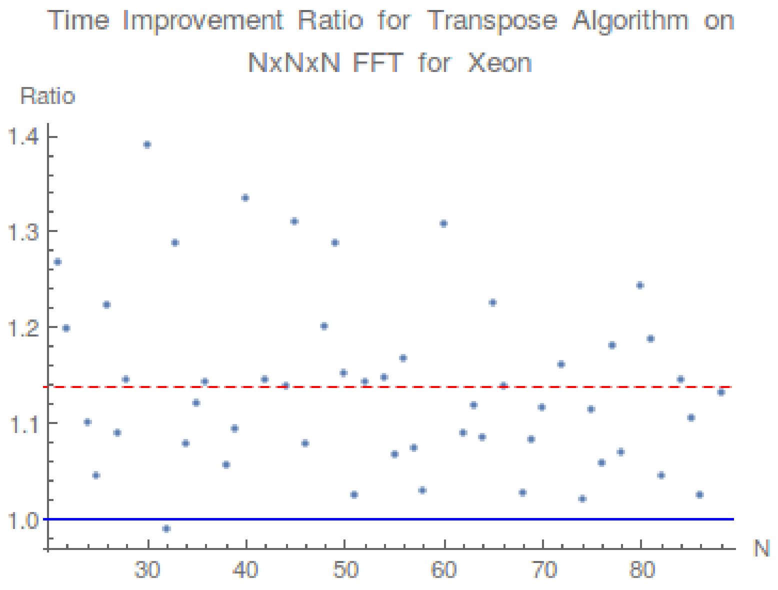
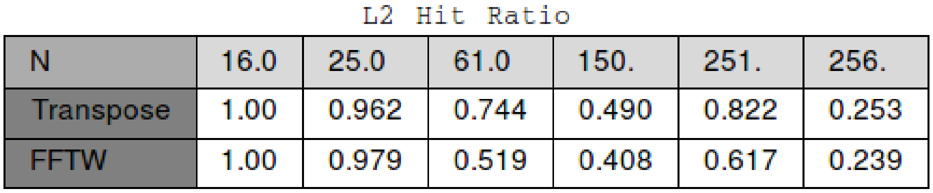


Figure 12: Time improvement ratio before and after introducing the transpose algorithm on the Xeon. An FFT was computed 1000 times, and the average time was taken. The legend is the same as for 11. The average was 13.8% improvement.

nets (figure 14), FFTs take up more computation time, so the savings are greater. On shallower nets, the effect is diminished (figure 15).

On the Xeon, an experiment analogous to figure 14 results in 3% improvement.



L2 Hit Ratio

N	16.0	25.0	61.0	150.	251.	256.
Transpose	1.00	0.962	0.744	0.490	0.822	0.253
FFTW	1.00	0.979	0.519	0.408	0.617	0.239

Figure 13: VTune-reported L2 hit ratio for forward and backward FFTs, run 10000 times for sizes below 100 and only 100 otherwise. After size 25, a cube and its output copy no longer both fit in the 128KB L2 cache of the Xeon (see figure 2).

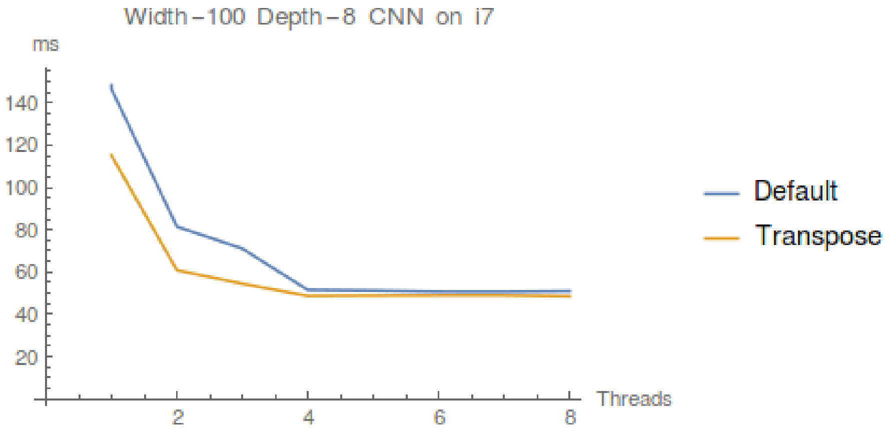


Figure 14: Average training iteration for 100-width, 8-depth CNN from 10 iterations overall. Average improvement in ratio of time is 14.3%.

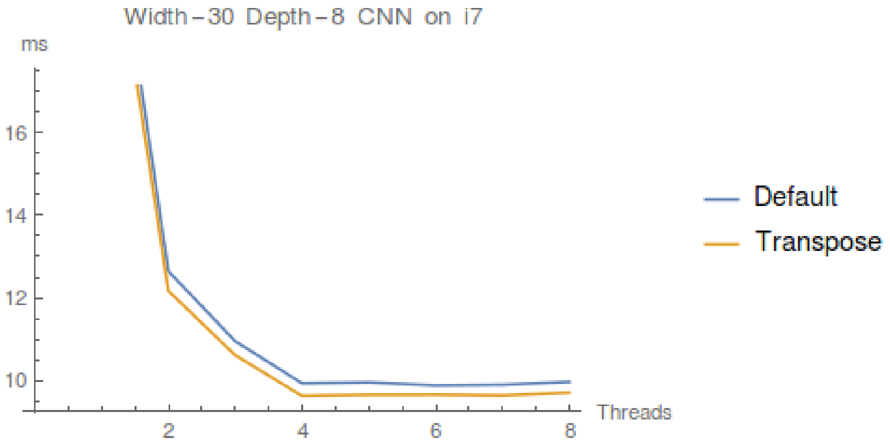


Figure 15: Average training iteration for 30-width, 8-depth CNN from 10 iterations overall. Average improvement in ratio of time is 3.06%.

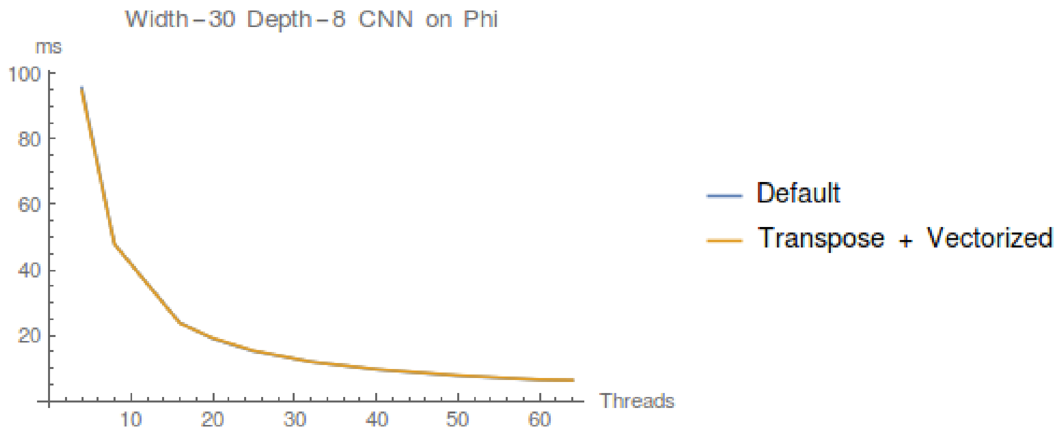


Figure 16: Even with the transpose algorithm to aid in the cache performance for the vectorized Xeon Phis, there are no improvement gains (in fact, an average slow down of 0.4%, most likely due to variance). This is still an improvement over the vectorization-only slow down.

Conclusion

In summary, focusing on vectorization improved performance by over 50% on certain prime cases and architectures, and introducing the transpose algorithm had a positive effect overall, which is increased for wider networks that are dominated by FFT time. A 10% to 15% average improvement in FFTs translated to a 3% to 14% improvement in the overall neural network.

The very modest speed gains are due to cache behaviour that isn't fully tuned. Exploring Intel's MKL package, which outperforms FFTW on larger-sized FFTs, one finds extremely high L2 cache hit ratio. Note MKL tends to outperform even the transpose algorithm in hit count for the larger sizes (figure 17).

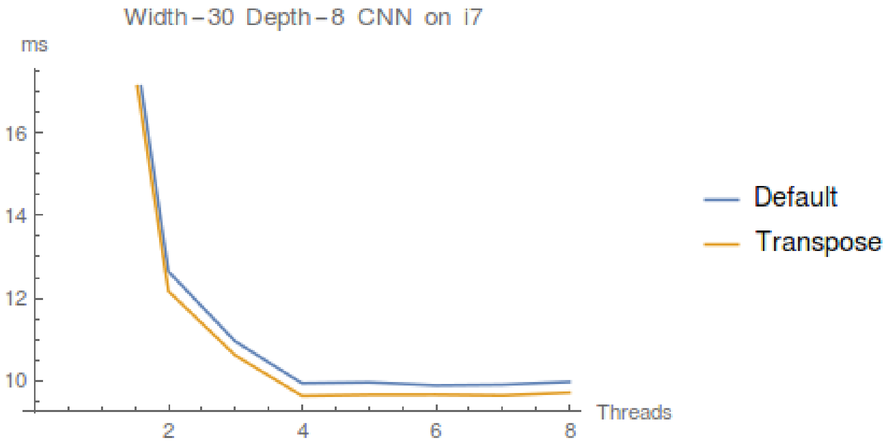


Figure 17: The same L2 hit diagram from before (figure 13, with Intel MKL performance for comparison.

In fact, observing the relative performance for FFTW and MKL, one notices that the two packages are complimentary, in figure 18.

This suggests a promising policy which trades off between the various packages. However, because the idea was not sufficiently original in itself, it was not pursued in this work.

However, there are other interesting directions for improvement as well. First, the use of hyperthreads across all three architectures shows performance degradation (not just a lack of scalability). This is likely due to the low-level caches being split amongst the threads, which results in problematic thrashing. A more intelligent scheduler would dispatch hyperthreads only

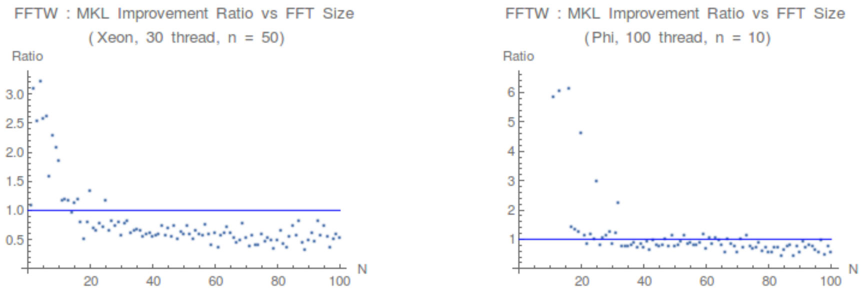


Figure 18: At around $N = 25$, on both the Xeon and Phi platforms, FFTW and MKL switch roles as fastest package.

for computations that rely on the same data. Because a convolutional neural net's convolution operations work on independent kernels, such parallelization can likely only be exploited within the FFT computation for the kernel itself.

Additionally, the FFT computation time can also be reduced by avoiding invocation of Rader's algorithm at all. FFTW and MKL both benefit from operation on composite numbers that are powers of 2,3,5 (FFTW also handles 7, 11 and 13 less well) [16] [14]. While it is easy to pad a number to the nearest power of two, it is wasteful (and unacceptable on the memory-constrained Phi platform). However, using all the available small-prime radices of FFTW, one can find much smaller composite numbers to pad to because multiples of the three radices are much more dense than just powers of two. An FFT of a 0-padded signal is not equivalent to a truncated FFT, however, for the operations that the CNN requires, once the inverse FFT is truncated, the result is mathematically equivalent to the original prime-sized FFT.

The code for the transpose algorithm is available on Github (`https://github.com/vlad17/znn-release`), as is the vectorized FFTW code (`https://github.com/vlad17/fftw3`). Testing scripts are available upon request from the author (vyf@princeton.edu).

Acknowledgements

First and foremost, I would like to acknowledge my adviser. Prof. Kai Li has suggested to me this challenging problem in an area of modern research and driven me to explore creative solutions in a difficult space. Furthermore, I am thankful for his frequent advice, availability, insights, and thoughtful responses throughout the process as my adviser.

I would like to thank Dr. Ben Singer for his frequent assistance with cluster setup, as well as the Princeton Neuroscience Institute itself for providing the resources on the `metacortex` cluster to allow for this investigation.

Yida Wang and Intel support also provided help with using VTune. In addition, the PNI resources were donated by the Intel Corporation for Prof. Kai Li.

References

[1] Best Practice Guide: Intel Xeon Phi v1.1. http://www.prace-ri.eu/. [Online]. Available: http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Intel-Xeon-Phi.pdf

[2] Intel core i7-5600u processor (4m cache, up to 3.20 ghz). https://software.intel.com/. [Online]. Available: http://ark.intel.com/products/85215/Intel-Core-i7-5600U-Processor-4M-Cache-up-to-3_20-GHz

[3] Intel® xeon® processor e5-2670 (20m cache, 2.60 ghz, 8.00 gt/s intel® qpi). https://software.intel.com/. [Online]. Available: http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI

[4] Optimization and Performance Tuning for Intel Xeon Phi Coprocessors. https://software.intel.com/. [Online]. Available: https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization

[5] Parallel Programming and Optimization with Intel Xeon Phi Coprocessors. http://www.umbc.edu/. [Online]. Available: http://www.umbc.edu/hpcf/user-resources/Colfax-training.pdf

[6] Specification for Xeon Phi (Knight's Landing). https://software.intel.com/. [Online]. Available: https://software.intel.com/sites/default/files/managed/e9/b5/Knights-Corner-is-your-path-to-Knights-Landing.pdf

[7] *9th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, ISBI 2012, May 2-5, 2012, Barcelona, Spain, Proceedings*. IEEE, 2012. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6227656

[8] D. H. Bailey, "Ffts in external of hierarchical memory," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. ACM, 1989, pp. 234–242.

[9] S. Chintala. Convnet Benchmarks. https://github.com/soumith/convnet-benchmarks. Available: https://github.com/soumith/convnet-benchmarks

[10] D. Ciresan *et al.*, "Deep neural networks segment neuronal membranes in electron microscopy images," in *Advances in neural information processing systems*, 2012, pp. 2843–2851.

[11] D. C. Ciresan *et al.*, "Deep neural networks segment neuronal membranes in electron microscopy images," in *NIPS*, 2012, pp. 2852–2860.

[12] Cooley, James W. and Tukey, John W., "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, pp. 297–301, 1965.

[13] I. Cooperation, "Intel cilk plus language specification," 2011.

[14] I. Corporation. (2010) Fft optimized radices. Available: http://scc.qibebt.cas.cn/docs/library/Intel%20MKL/2011/mkl_userguide/MKL_UG_managing_performance/FFT_Optimized_Radices.htm

[15] J. Fang *et al.*, "An empirical study of intel xeon phi," *arXiv preprint arXiv:1310.5842*, 2013.

[16] M. Frigo and S. G. Johnson, "Fftw user's manual," *Massachusetts Institute of Technology*, 1999.

[17] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[18] M. Frigo *et al.*, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 285–297.

[19] A. Karpathy *et al.*, "Large-scale video classification with convolutional neural networks," in *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*. IEEE, 2014, pp. 1725–1732.

[20] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014. Available: http://arxiv.org/abs/1404.5997

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[22] B. Z. Kumar Aatish. (2015, Dec.) Three dimensional fast fourier transform cuda implementation. Available: http://cseweb.ucsd.edu/~baden/classes/Exemplars/cse260_fa12/3DFFT.pdf

[23] Y. LeCun *et al.*, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, November 1998.

[24] K. Lee *et al.*, "Recursive Training of 2D-3D Convolutional Networks for Neuronal Boundary Detection," *CoRR*, vol. abs/1508.04843, 2015. Available: http://arxiv.org/abs/1508.04843

[25] Y.-Q. Liu *et al.*, "Memory efficient two-pass 3d fft algorithm for intel® xeon phitm coprocessor," *Journal of Computer Science and Technology*, vol. 29, no. 6, pp. 989–1002, 2014.

[26] Mathieu, Michael and Henaff, Mikael and LeCun, Yann, "Fast Training of Convolutional Networks through FFTs," in *International Conference on Learning Representations (ICLR2014)*. CBLS, April 2014. Available: http://openreview.net/document/aa6ab717-ca19-47e1-a958-823b9a106ca9

[27] A. D. Robison, "Cilk plus: Language support for thread and vector parallelism," *Talk at HP-CAST*, vol. 18, 2012.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, p. 3, 1988.

[29] R. Schulz, "3d fft with 2d decomposition," *CS project report http://cmb. ornl. gov/Members/z8g/csproject-report. pdf*, 2008.

[30] J. Seward, N. Nethercote, and J. Weidendorfer, *Valgrind 3.3-Advanced Debugging and Profiling for GNU/Linux applications*. Network Theory Ltd., 2008.

[31] D. Takahashi, "An implementation of parallel 1-d fft using sse3 instructions on dual-core processors," in *Applied Parallel Computing. State of the Art in Scientific Computing*. Springer, 2007, pp. 1178–1187.

[32] Torch. `cunn`. https://github.com/torch/cunn. Available: https://github.com/torch/cunn

[33] Torch maintainers. Torch. https://github.com/torch/torch7. Available: https://github.com/torch/torch7

[34] Y. Wang *et al.*, "Optimizing Full Correlation Matrix Analysis of fMRI Data on Intel Xeon Phi Coprocessors."

[35] A. Zlateski, personal correspondence, 2015.

[36] A. Zlateski, K. Lee, and H. S. Seung, "ZNN - Fast and Scalable Algorithm for 3D ConvNets on Multi-Core and Many-Core Shared Memory Machines," *Under Consideration*.