Compile–time Optimized and Statically Scheduled N–D ConvNet Primitives for Multi–Core and Many–Core (Xeon Phi) CPUs

Aleksandar Zlateski*, H Sebastian Seung†
Princeton Neuroscience Institute
and Computer Science Department
Princeton University
Princeton, NJ 08540 USA
*zlateski@princeton.edu, †sseung@princeton.edu

***Abstract*—With the increasing computation power of hardware, convolutional networks (ConvNets), branded as deep–learning, became the most popular approach to computer vision. Vast majority of the computation performed during both training and using trained ConvNet consist of convolutions with small kernels. Being capable of more FLOPs/s, GPUs have been a hardware of choice**

As the CPUs are closing the FLOPs gap, it is important to have an efficient CPU algorithm. We propose a novel parrallel and vectorized algorithm for the convolutional layers. Our algorithm is designed to be agnostic to both the ConvNet architecture and the CPU capabilities and cache sizes.

To achieve high utilization, we assume that the network architecture will be known at compile–time. Our serial algorithm divides the computation into small sub–tasks designed to be easily optimized by the compiler so that they utilize the FMA hardware and the register file. The sub–tasks are executed in an order that maximizes cache reuse. We parallelize the algorithm by statically scheduling tasks to be executed by each core. Our novel compile–time recursive scheduling algorithm is capable of dividing the computation evenly among an arbitrary number of cores, regardless of the network architecture. It introduces zero runtime overhead and minimal synchronization overhead.

Our serial primitives achieve very high utilization (75-95%) of the hardware, while our parallel algorithm attains 50-90% utilization on 64+ core machines.

We benchmark our primitives on the state of the art ConvNets. For object detection networks we perform marginally better on Xeon and marginally worse on KNL. For segmentation our approach is superior. Finally, for 3D networks we show competitive performances to the latest cuDNN and latest GPU hardware, even though the GPU has higher FLOPS.

Ideas 1) Compile time optimized, leveraging the compiler for optimal serial code

2) Static scheduling

Points 1) Portable + future generations

2) Support N-D primitives

3) Similar performances regardless of the layer shape

4) Depending on the layer, results are either competative, or superior to the state of the art alternatives (MKL2017)

5) Cache oblivious

I. Introduction

ConvNets had become... popular seung ►*intro about convnets*◄ The argument of using CPUs or GPUs for deep learning had reached almost religious proportions. The GPUs, which are capable of more FLOPs/s seem to win the battle by a large margin – disproportional to the FLOPs/s available to the two architectures. We find this absurd, as the CPUs have been around for much longer, and the CPU programming model changes very little from a generation to the next. On the other side, GPUs seem to change a lot.

As the CPUs are closing the gap in terms of FLOPs/s performance. Xeon Phi Knights Landing chips are capable of up to 6TFLOPs/s []. Upcoming Skylake server CPU's will be packing 3.2 TFLOPs/s per single chip, delivering up to 25.8 TFLOPs/s on an 8-way configuration []. We believe that the main battle should be in harvesting the computational power of the hardware. We are surprised to see that the GPU implementations have higher utilization of the % FLOPs/s available, compared to the CPU.

Reducing convolution to matrix multiplication has been a popular approach due to the availability of highly efficient matrix multiplication routines. This approach has, however, both a computational and memory overhead due to the requirement of creating in–memory matrices. Caffe con Troll (CCT) [1] uses blas to perform convolutions – we will compare us to them.

Initially, cuDNN [2] took that approach, however, it has since implemented more efficient direct convolution algorithms.

Convolutional layers of a ConvNet are computationally dominating both training and inference. The forward and backward pass involve a convolution of an image with a small kernel. Typically "valid" dense convolution, possibly with strides is performed. More advanced types of convolutions can decomposed to a series of dense convolutions [3].

To achieve high utilization of the CPU, one

We propose a new approach. We harvest the power of C++ templates to create compile time primitives by harvesting the compiler. Given the specifications of the CPU, our meta–algorithms optimize and vectorize single threaded code, as well as generate static scheduling for parallelizing the computation over multiple cores.

We show that our approach can utilize extremely high percentage of the available FLOPS regardless of the CPU

generation, as well as have near linear scalability over the number of cores available. The performances of our algorithms are competitive with the state of the art hand optimized code for specific architectures. Our code yields better performances on the KNL compared to the state of the art GPU implementations for 3D convolutional networks, even though KNL has less FLOPS/s available.

We further demonstrate the power of our approach for inference only computation.

Even though we provide an efficient, publicly available, open source implementation for popular ConvNet primitives, the goal of this paper is to point to the right direction for solving the problem, rather than giving the final "optimized" implementation.

Efficient algorithms for training and inference of ConvNets with larger kernels have been introduced in [4], [5], however the evidence suggests that ConvNets with smaller kernels, and larger number of layers perform better on nearly all problems. (Cite).

A. GPUs vs CPUs

GPUs have streaming multiprocessors, while the CPUs have in–order, memory coherent cores.

GPUs model is so called SIMT (single instruction multiple threads)... dwell on it..

CPUs on the other side have a complex structure of many cores and virtual cores, many cache layers and each core capable of performing SIMD instructions on up to 16 floating point numbers.

In this paper we are not trying to answer the GPU vs CPU question, but rather propose an algorithm and implementation for ConvNet primitives that can utilize very high percentage of the theoretical peak performance of both many–core (Xeon Phi) and multi–core (Xeon) CPUs. As shown later, our algorithm performs well on multi–chip systems as well.

B. Mulit–core vs Many–core

Explain the differences NUMA, caches, etc..

C. Motivation

Modern CPUs can achieve high performances under very special circumstances. Mainly, each core of modern CPUs has up to two FMA vector units that are capable of performing $2S$ floating point instructions per cycle. Here, S is the width of the vector register (how many floating point numbers fit inside the register). The peak performace of a single core is then reached when, in each cycle, all available FMA units perform a fused multiply–add operation whith the input stored in the register file. Additionally, each CPU has a different latency of the FMA units. This means that the result of the computation is not immidiately available, but rather after c cycles, where c is the latency of the unit. To fully utilize the n FMA units, a program must continiously issue FMA instructions such that no nc consecutive instructions are dependent (a result of one instruction is an input for some other one). Modern compilers are aware of these limitations, and will try to assemble the code in the most optimal way.

Loading data from memory to a register introduces an additional overhead. Thus, to achieve high performance, one must minimize such data transfers and re–use data in the register file as much as possible. Furthermore, fast loading of data from memory requires additional constraints. Fast loading S values into a vector register requires the values to be consecutive and properly aligned in memory.

Efficiently utilizing Multi–core and many–core induce an additional constraint. In order to fully utilize all available cores, they must perform independent computation and minimize synchronization. A core shouldn't wait for a result computed by another core.

Because of these constraints, the naive way of computing convolutions is very inefficient. Designing an algorithm that follows all the constraints is challenging. To design such algorithm, we are motivated by the following two principles. First, we divide computation into small sub–tasks. We provide a relatively simple implementation for each task so that the compiler can generate optimal machine code. Secondly, the computation will be evenly distributed among the available cores, such that each core does an equal amount of work. This minimal synchronization is necessary and each core can start and ends the computation at the same time.

II. Convolutional layers

We begin by describing the computation performed during the forward, backward and the update pass of a convolutional layer.

In the forward pass of a convolutional layer a tuple of f images are transformed into another tuple of f' images. We want to process a batch of S inputs to yield a batch of S outputs, via

$$O_{s,j} = \sum_{i=1}^{f} w_{ji} * I_{s,i}$$

for $1 \leq s \leq S$ and $1 \leq j \leq f'$. Here $I_{s,i}$ is the i^{th} image of the s^{th} input in the batch, and $O_{s,j}$ is the j^{th} image of the s^{th} output in the batch, and w_{ji} is the kernel from the i^{th} image in an input tuple to the j^{th} image in an output tuple.

We will assume 3D images and kernels. If $I_{s,i}$ has size $\vec{n} = \langle n_x, n_y, n_z \rangle$ and w_{ji} has size $\vec{k} = \langle k_x, k_y, k_z \rangle$, then we can regard I as a 5D tensor of size $S \times f \times n_x \times n_y \times n_z$, w as a 5D tensor of size $f' \times f \times k_x \times k_y \times k_z$, and O as a 5D tensor of size $S \times f' \times n'_x \times n'_y \times n'_z$, where $\vec{n}' = \vec{n} - \vec{k} + \vec{1}$.

We will refer to the sizes of the 5D tensors I and O as input and output shape, respectively. The relationship between input shape and output shape depends on kernel size as in Table **??**.

The forward pass processes

A. Data layout

Both the input and output of both the forward pass and backward pass of an N–dimensional convolutional layer can be represented as $(N+2)$ dimensional tensor. The pixel at $(x_1, x_2, \ldots, x_N)$ of c^{th} image in the batch b is located at $(b, c, x_1, x_2, \ldots, x_N)$

Similarly, the kernels can also be represented as $(N+2)$ dimensional tensor, where the $(x_1, x_2, \ldots, x_N)$ element of the kernel

To efficiently use FMA instructions we use data layout as proposed in [6], [7], except that we generalize it for N–dimensional case.

Instead of using $(N+2)$ dimensional tensor for the input/output images we use a $(N+3)$ dimensional tensor such that the element $(b, c, x_1, x_2, \ldots, x_N)$ of the original tensor is mapped to an element $(b, \lfloor c/S \rfloor, x_1, x_2, \ldots, x_N, c \mod S + 1)$. This new tensor size of $B \times \lfloor C/S \rfloor \times X_1 \times X_2 \times \cdots \times X_N \times S$.

S denotes the length of the CPU's SIMD register in terms of number of floats it can store. This is one of very few parameters required to specify when compiling our code for different CPUs. In the case of Haswell and Skylake S equals to 8 and the Knights Landing has $S = 16$. Note that when C is not divisible by S, the new tensor will have more elements, and thus require more memory than the original tensor. However, nearly all modern ConvNets **Aleks** ▶*Cite alex-net, googlelenet, oxfordnet*◀ have C as a multiple of 16.

$1/2$ aleksa $\mod$

$$O_{b,j} = \sum_{i=1}^{f} I_{b,i} * w_{ji}$$

And for the backward pass

$$\frac{\partial L}{\partial I_{b,i}} = \sum_{j=1}^{f'} \frac{\partial L}{\partial O_{b,j}} \star w_{ji}$$

Update

$$\frac{\partial L}{\partial w_{j,i}} = \sum_{b=1}^{B} I_{b,i} * \frac{\partial L}{\partial O_{b,j}}$$

III. Static scheduling

The common underlying theme of our forward/backward propagation as well as update phase algorithms is static scheduling of the work for each available core. This allows us to minimize the synchronization overhead by equally distributing the work and using a one or two fork–join calls.

In order to fully utilize N cores, we need to divide the work into exactly N independent parts that perform the same number of operations. Ideally, we would like all the cores to execute the same code and have the same memory access patterns. This approach relies on having exclusive access to all the cores. This is reasonable as one can limit the number of cores used for the ConvNet computation, while leaving some cores to the system.

There are multiple reasons why we choose this approach over the alternative, dynamic scheduling. To efficiently utilize many cores with a dynamic scheduling approach we would have to split the problem into many fine granularity tasks. As the number of threads increases, maintaining a synchronized global queue of available tasks becomes more expensive. Having multiple local queues and a work–stealing dynamic scheduler [8], [9] can only partially lower the synchronization overhead. Additionally, with a dynamic scheduling we don't have control over the memory access patterns of each of the cores, as any core can possibly execute any task. This can yield a large overhead due to cache misses, especially on multi–chip machines, where using L3 cache is crucial due to NUMA [1].

A. Implementation details

We provide custom fork–join primitive. The main thread of the program is pinned to the core 0 as described in [10]. Additional $N-1$ threads are spawned and pinned to cores 1 to $N-1$. The main thread continues with the execution of the program, while the other threads block on a single barrier. On a CPU with C cores and H hyper–threads, h^{th} hyper–thread of the c^{th} core executes the thread number $c \times H + h$, Note that this differs from the standard unix thread assignment where the same core/hyper–thread is assigned to the system thread with the id of $h * H + c$. We choose this thread assignment in order to have local threads execute on local, and possibly the same physical core. For multi–chip system, local threads will be assigned to the same physical chip.

On fork, threads 1 to $N-1$ are unblocked to execute tasks 1 to $N-1$, and the calling thread executes the task 0. After the execution of the task, each thread waits on a single barrier. When all threads complete their task, the main thread continues with the program execution, while the remaining $N-1$ threads block. Our implementation uses `pthreads`.

IV. Forward and backward propagation

There are two main differences between the forward and backward pass computation. While the forward pass computes convolutions, the backward pass computes cross–correlations. Performing a cross–correlation is equivalent to performing a convolution with kernel that is reflected along all dimensions. Secondly, during the forward pass the convolution is performed only for locations where the kernel is fully contained inside the image[2]. Thus, the output image has dimensions smaller or equal to the input image. In the backward pass, `full` convolution/cross–correlation is performed. Image size increases because an output voxel

[1] Non–uniform memory access

[2] This is known as a `valid` convolution in MATLAB.

exists whenever the sliding window has some overlap with the input image.

Given an algorithm for the forward pass, a backward pass can be implemented by (1) reflecting the kernels along all directions, thus converting cross–correlation to convolution, and (2) zero–padding the input image such that `valid` convolution becomes `full` convolution.

We propose an algorithm for the forward pass that can handle an arbitrary zero padding of input image, thus the same algorithm can be used for the backward pass. We assume that two copies of kernels are stored - original and reflected. Both sets of kernels will be updated during the update phase. This is reasonable because the memory required for kernels is typically much smaller than the memory required for the images. Updating both sets of kernels instead of just one introduces very little overhead for the update phase, while allowing the reuse of the same algorithm for both the forward and backward computation.

We will refer to the algorithm as the fwd-bwd algorithm. Also, we'll refer to both the input/output images of the forward pass and the input/output gradients of the backward pass as *input/output images*. For simplicity, we will describe the details of the algorithm for the special case of 3D images. Higher dimensions require adding an additional for loop in all of our primitives.

A. Algorithm overview

The result the computation of the fwd-bwd algorithm is B sets of F (or F') N–dimensional images that can be represented as a $N+2$ dimensional tensor. Computing the value of each element of the tensor is independent. We divide this tensor into smaller tensor by slicing it along different directions.

B. Problem subdivision

To compute all the values of the output tensor using T threads, we would like to divide the tensor into T sub–tensors of equal size. Such sub–division is not guaranteed to exist.

To equalize the amount of work done by each thread, we propose a recursive subdivision approach. We first divide the problem into T equal sub–tensors, with a possible remainder. Each of the threads computes the values of a single such sub–tensor. The remaining sub–tensor is then computed in a similar fashion.

An example of such sub–division is shown on Fig 1. Note that each thread ends up computing the same amount of output voxels. Furthermore, at each stage, each thread is solving the same *Problem*, just with different data (SIMD).

In each stage of the computation, each of the threads computes the values of the assigned sub–tensor in a serial fashion. For that we need an efficient serial algorithm that can compute the values of an arbitrary sub–tensor. It is improtatnt to first understand our serial algorithm, as it will introduce some restrictions on the problem sub–division.

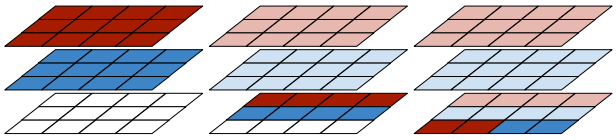

Figure 1: Computing the values of $F' = 3$ channels of 2D images of size 4×3 using $T = 2$ threads. The three columns represent three stages. The dark blue/red represent the values computed by the cores 1/2 during that stage.

C. Serial algorithm

Our serial algorithm computes $\mathcal{B} \leq B$ sets of $\mathcal{F}' \leq F'$ images each of size $\mathcal{D} \times \mathcal{H} \times \mathcal{W}$ ($\mathcal{D} \leq D, \mathcal{H} \leq H, \mathcal{W} \leq W$).

We take a divide and conquer approach to the serial algorithm. We first divide the algorithm into primitives that performs S^2 convolutions on S input images, accumulating the result into S output images. These primitives are further divided into sub–primitives that compute a small number of pixels of each of the S output images.

We first describe the most basic building block of our algorithm. The main goal of this primitive is to efficiently utilize the vector units as well as $L1$ cache. We design the primitive so that it consists mainly of fused multiply–add instructions where two of the input arguments are stored in registers, and the third one is in $L1$ cache. For the case of KNL, a single instructions can be used to perform such operation, whereas with AVX2 capable CPUs we need to use a set of auxiliary registers to first load the data from $L1$. We do not implement the two different cases, instead we let the compiler optimize the code for the specific architecture.

Algorithm 1 Serial forward.

FORWARDTASK$\langle B, O_D, O_H, O_W, W_D, W_H, W_W \rangle (i, w, o, b)$

```
1  simd register oreg[D][H][W]
2  simd register wreg
3  if B // Static if
4      oreg[:][:][:][:] = b // Load bias
5  else
6      oreg[:][:][:][:] = LOAD(o[:][:][:][:])
7  end if
8  for k_d = 0 to W_D − 1
9      for k_h = 0 to W_H − 1
10         for k_w = 0 to W_W − 1
11             for s = 0 to simd-width − 1
12                 wreg = w[k_d][k_h][k_w][s][:]
13                 unrolled for d = 0 to O_D − 1
14                     unrolled for h = 0 to O_H − 1
15                         unrolled for w = 0 to O_W − 1
16                             oreg[d][h][w] = oreg[d][h][w]
17                             +wreg * i[d + k_d][h + k_h][w + k_w][s]
```

The algorithm that computes a S sub–images of size $\mathcal{D} \times \mathcal{H} \times \mathcal{W}$ is given on 1. It consists of three main stages. First, it loads the current values for the pixel values to the register file, then it performs a convolution with the corresponding kernels and input images, and finally stores the results back to memory.

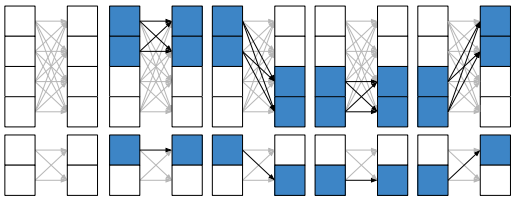

Figure 2: Cache oblivious compuitation.

The convolution loop first goes over all kernel values, loading each of them into a register. That value is then multiplied with the appropriate pixels from S input images and accumulated to the registers containing the output set of pixels.

The innermost loop over S ensures efficient reuse of $L1$ cache. Mainly, it loops over $D \times H \times W$ pixels of S images. As the L1 cache is larger than the number of registers times S, we expect that the consequent loops over S will be guaranteed to have the value in L1 cache.

This will however depend on the shape of $D \times H \times W$. The main restriction of the shape is that it has to fit into the register file. For the case of KNL we set the max size of elements to be 31 (as we need an extra register for the kernel values, and we have total of 32 registers). For the case of AVX2, we set it to be at most 7. This is because we need as many auxiliary registers to load input image pixels, as well as an extra for the kernel values (and we have 16 registers in total).

We also prefer higher values for the least significant dimensions (W in this case). There are two main for this design choice which we discuss in sections.... They are hardware prefetching, cache associativity conflicts.

The actual implementation of the primitive uses regular variables and it expect the compiler to assemble the code that uses the register file, as well as to optimally unroll the innermost loops. It turns out that both ICC and GCC produce the same optimal code as expected for AVX2, whereas only ICC produces optimal code for AVX512. For AVX512, GCC fails to use the braadcast-fmadd instructions.

We combined these, lowest level, primitives to compute S^2 convolutions of S input images by combining the primitives described above. Each of the S output images will be split into blocks of size at most $O_D \times O_H \times O_W$, and the computation will be performed by computing one block at the time. Note that not all the blocks will have the same size (in the case when the output image size is not divisible by the register blocking size). By utilizing the C++'s template programming we will have a separate functions for computing blocks of different sizes.

We prefer computing the blocks adjacent to the least significant dimension (in order to utilize the L2 cache and hardware prefetching **Aleks** ►*elaborate*◄.

D. Cache oblivious computation

For each set of

Given the primitive described above, we attempt to compute all $B \times \beta S \times D \times H \times W$ values in the following way.

Given the primitives that perform the forward pass on S input images and accumulate the result to S output images, given S^2 convolution kernels, we design a cache oblivious algorithm that performs the forward pass of an arbitrary number of input/output images.

E. Parallel execution

Our strategy for is relatively simple and is created during compile time.

The main idea behind our static scheduling algorithm is to have all threads execute the same code, thus having the same number of operations/FLOPS to execute.

To achieve this, we recursively divide the problem and assign a set of threads to execute a sub–problem. To divide the problem among T threads, we consider the smallest prime P that divides T. We divide the problem into P equal sub–problems, and recursively schedule the sub–problems to each of T/P sub-threads. If the problem can not be divided in exactly P parts, we divide it into P parts + a smaller remainder. The remainder is then also recursively scheduled on all T threads.

All scheduling and code generation is done during compile–time with the magic of C++ templates. Thus each sub–problem get's optimally compiled and optimized.

The obvious advantage of such approach is that, by executing the exact same code, threads that share any level of cache, can leverage the caching of the instructions. Other advantages are non–obvious, to understand them, we need to see how the problem gets recursively divided. For simplicity, we will assume that the number of available threads is a power of 2, the same approach can be taken for an arbitrary number of available threads. However, in practice, best divisions can be obtained if T is a product of small primes. Our implementation limits T to be product of $2, 3, 5$ and 7.

F. Serial algorithm

In this section we describe both the serial vectorized approach to computing the forward and backward pass as well as a parallel algorithm that combines these serial primitives.

As described before, we keep two copies of the kernels in two different memory layouts, one suitable for both the forward and backward pass computation. Our algorithm is supports implicit zero padding, hence both the forward and backward pass can be computed using the same algorithm.

G. Cache, etc...

In order to be portable, our algorithm doesn't assume any specific size of available caches. However it assumes that small $L1$ data cache is present, able to fit $2 \times S \times R$ floats.

R represents the number of available SIMD registers. This is true for all commercially available CPUs.

Also, we assume presence of higher levels of cache (L2,L3), however, our algorithm is oblivious to the size of the cache. Having more cache will, however improve the performances of the algorithm.

V. Update phase

As in the fwd-bwd case, we will describe the details of the algorithm for the case of 3D images.

A. Serial algorithm

Similarly as in the fwd-bwd case, we first divide the algorithm into primitives that compute S^2 convolutions of S input images with S gradients.

We implement the following algorithm described in Algorithm 2 that computes a convolution of an R_F images of size $R_D \times R_H \times (W + R_W - 1)$ by S images of size $1 \times 1 \times W$ separately, to produce and accumulate the results of $R_F \times S$ images of size $R_D \times R_W \times R_H$.

Algorithm 2 Serial update subtask.

```
UPDATE-SUBTASK⟨R_D, R_H, R_W, W, R_F⟩(a, b, r)
 1  simd register oreg[R_D][R_H][R_W][R_F]
 2  simd register wreg
 3  oreg[:][:][:][:][:] = LOAD(r[:][:][:][:][:])
 4  for i = 0 to W − 1 // Partially unrolled
 5      wreg = LOAD(b[1][1][w][:])
 6      PREFETCH-TO-L1(b[1][1][w + 1][:])
 7      for d = 0 to R_D − 1 // Fully unrolled
 8          for h = 0 to R_H − 1 // Fully unrolled
 9              for w = 0 to R_W − 1 // Fully unrolled
10                  PREFETCH-TO-L1(a[d][h][w + i + 1][:])
11                  for f = 0 to R_F − 1 // Fully unrolled
12                      oreg[d][h][w][f] = FMADD(
13                          wreg,
14                          EXLOAD(a[d][h][w + i][f]),
15                          oreg[d][h][w][f])
16                  end for f
17              end for w
18          end for h
19      end for d
20  end for i
21  r[:][:][:][:][:] = STORE(oreg[:][:][:][:][:])
```

The algorithm relies on the compiler to use register file, and unroll loops. In order for that to be possible, the $oreg$ and $wreg$ variables have to fit in the register file. That means that $R_D \times R_H \times R_W \times R_F + 1 \leq R$, where R is the number of vector registers available to the system. Note how most of the computation involves FMADD instructions with the arguments either in the register file, or in $L1$ cache.

Performing a convolution of S images of size $K_D \times K_H \times K_W \times (W + K_W - 1)$ with S images of size $1 \times 1 \times W$ can be performed by computing $R_D \times R_H \times R_W$ results of $S \times R_F$ results using the algorithm described above. We prefer decompositions for which the values of $R_D \times R_H \times R_W \times R_F$ are large, but smaller than R. Out of different

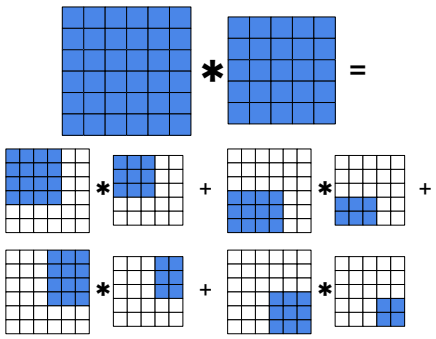

Figure 3: Decomposing a convolution.

combinations we prefer ones for which R_F is maximized, then R_W is maximized, and so on.

We would like to use at least half of the available registers, which can be always accomplished by choosing $R_F = S$, as for all architectures $S = R/2$. However, better solutions might exist. We find the best solution during compile time using C++ meta programming. The solution has to divide the computed values evenly (into subsets of the same size). For example, on AVX512 machine, where $R = 32$, $S = 16$, for a kernel of size $2 \times 3 \times 3$ (VD2D3D) we get a decomposition into $R_D = R_H = 1$, $R_W = 3$ and $R_F = 8$. This uses 24 of the available registers.

The computation is performed by sliding $R_D \times R_H \times R_W \times R_F$ first over the output images, then the least significant dimension, and so on.

Algorithm 3 Serial update task.

```
UPDATE-TASK⟨I_D, I_H, I_W, O_D, O_H, O_W⟩(a, b, r)
 1  ⟨K_D, K_H, K_W⟩ = ⟨I_D − O_D + 1, I_H − O_H + 1, I_W − O_W + 1⟩
 2  ⟨R_D, R_H, R_W, R_F⟩ = DECOMPOSE(K_D, K_H, K_W, S)
 3  for d = 0 to O_D − 1
 4      for h = 0 to O_H − 1
 5          for r_d = 0 to K_D/R_D − 1 by K_D/R_D
 6              for r_h = 0 to K_H/R_H − 1 by K_H/R_H
 7                  for r_w = 0 to K_W/R_W − 1 by K_W/R_W
 8                      for r_f = 0 to S/R_F − 1 by S/R_F
 9                          UPDATE-SUBTASK⟨R_D, R_H, R_W, O_W, R_F⟩(
10                              a[d + r_d, R_D][h + r_h, R_H][r_w, O_W][:],
11                              b[d][h][:][r_f, R_F],
12                              r[r_d, R_D][r_h, R_H][r_w, R_W][r_f, R_F][:])
13                      end for r_f
14                  end for r_w
15              end for r_h
16          end for r_d
17      end for h
18  end for d
```

Aleks ▶*Explain why and how*◀.

B. Parallel execution

VI. Optimized serial algorithm

In order to maximally utilize a single core we need to make sure that the CPU is using the FMA units all the time (when available) or performing the SIMD instructions.

We need

1) Cache blocking
2) Register blocking
3) Loop unrolling
4) FMA/SIMD utilization

VII. Static scheduler

Let $M = (I, D)$ be a serial method that executes instructions I on data D (e.g. I is a compiled method that receives a set of arguments D.

A serial program can be represented as an ordered list of methods $P = (M_0, M_1, \ldots, M_k)$. Our goal is to generate C parallel programs $P_0, P_1, \ldots, P_{C-1}$, such that for each $0 \leq i < j < C$ the following two claims are true.

1) Programs P_i and P_j do not write to the same memory location.
2) The first K methods of both P_i and P_j have the same instructions, where K equals the minimum of the number of procedures in both programs.

In addition, we would like to minimize the complexity of each program.

A. Meta–algorithm

FUNCTION$\langle B, F, F', I_d, I_h, I_w, W_d, W_h, W_w \rangle (i, o, w, b)$

Algorithm 4 Multi-core algorithm for a convolutional layer using direct convolution.

```
SCHEDULE⟨T, B, F, F', Ī, K̄⟩(t, i, o, w, b)
1  n̄' = n̄ − k̄ + 1̄
2  O = 5D-REAL-TENSOR(S, f', n'_x, n'_y, n'_z)
3  parallel for i = 0 to S − 1
4      parallel for j = 0 to f' − 1
5          for k = 0 to f − 1
6              O_{i,j} = O_{i,j} + CONVOLVE(I_{i,k}, w_{j,k})
7  FREE-MEMORY(I)
8  return O
```

VIII. Benchmarks

We benchmark state of the art 2D networks for both object–detection and semantical segmentation as well as 3D networks.

First, we demonstrate the near–linear scalability of our approach, regardless of the layer shape. We show that our approach has the same utilization for all of them.

Secondly, we compare the speed of our approach to alternative 2D primitives – specifically CcT, MKL–DNN and MKL–2017. We show that our approach is competitive, over–performing the next bet competitor by a small margin on Xeon and under–performing by a small margin on the KNL for 2D object–detection. However, for semantical segmentation, we greatly outperform all competitors on both Xeon and Xeon Phi.

Finally, we compare the speed of 3D networks to state of the art 3D primitives for the GPU.

Processor	GFLOPs/s
i7-6700K (Skylake)	512
4-way E7-8890v3 (Haswell)	5760
Xeon Phi 7210	4505.6
Titan X (Maxwell)	6600
Titan X (Pascal)	11000

Table I: Machines used for the benchmarks

A. Experimental setup

Describe the machines

B. Benchmarked networks

Describe the networks

1) VGG–A
2) U–Net
3) 3D network from https://arxiv.org/pdf/1412.0767.pdf

IX. Conclusion and remarks

A. Implementation details

Note that the ease of implementing the algorithm (under XXX lines of code, where XXX is very small **Aleks** ►*don't forget the CPPCON'17*◄). Publicly available. GCC beats ICC for regular Xeon by up to 5% (and takes much much less time). ICC beats GCC by 2x for KNL. We expect improved performances as the compilers get better.

Aleks ►*maybe try clang++, but probably not enough time for that*◄

Aleks ►*remember to mention that we don't measure cache hits because measuring on a single architecture wouldn't say anything about how general our approach is*◄

B. List of assumptions and design decisions

We prefer hardware pre–fetching over manual cache optimizations. For this reason we prefer larger input images.

We prefer blocking in the least significant directions to avoid associativity conflicts, this relieves us of data alignment issues. Hardware pre–fetching comes handy here as well.

We only assume small amount of L1 data cache.

We assume presence of L2 and possibly L3, but design the algorithm to be oblivious to the size of the cache available.

We assume presence of L1 instruction cache, and leverage the compiler to properly assemble the code. By having all threads run the same code we leverage possible sharing of L1 instruction and L2 cache (that stores the instructions).

We contribute to the compiler community, and expect our results to improve as AVX512 compilers improve.

This is just to see how much page estate will the references take [4], [5], [7], [11], [12], [13], [1], [2], [3], [6], [14], [10], [15], [16], [17], [18], [19], [20], [8], [21]

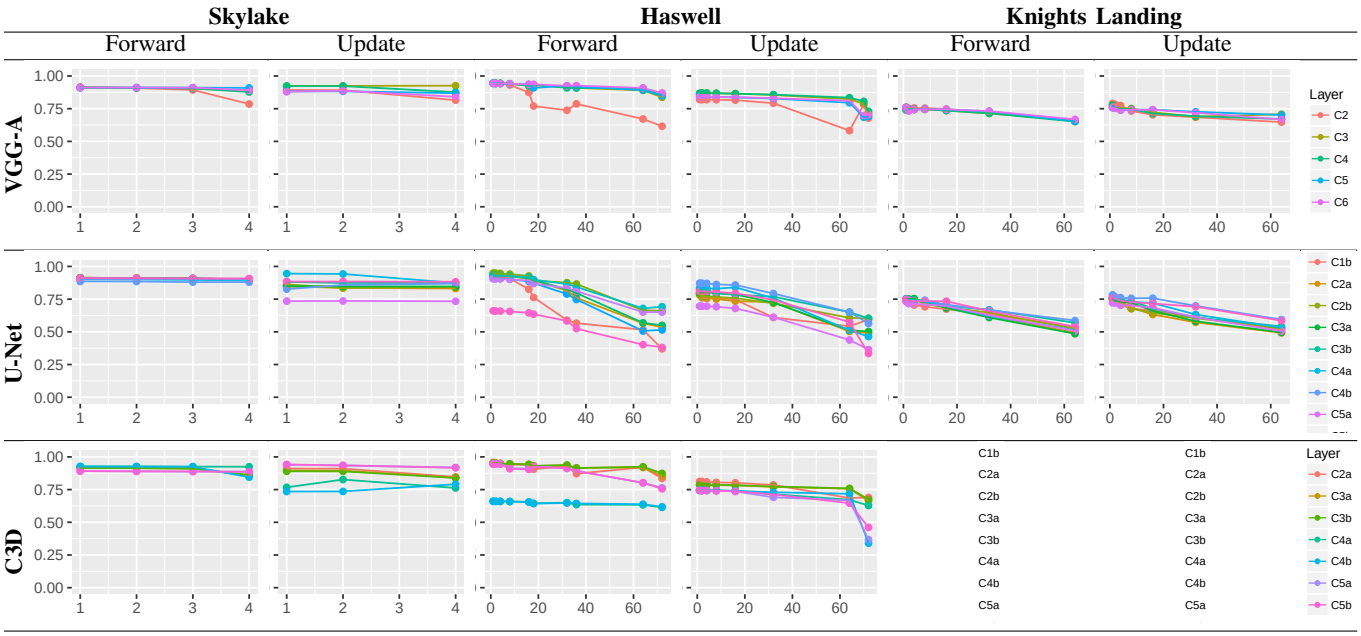

Figure 4: Some caption

References

[1] S. Hadjis, F. Abuzaid, C. Zhang, and C. R. C. con Troll, "Shallow ideas to speed up deep learning," in *Workshop on Data analytics in the Cloud (DanaC)*, 2015.

[2] 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.

[3] 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*, 2015, pp. 1–9.

[4] 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 *Proceedings of the 2016 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '16, 2016, pp. 801–811.

[5] ——, "Znni - maximizing the inference throughput of 3d convolutional networks on multi-core cpus and gpus," *arXiv preprint arXiv:1606.05688*, 2016.

[6] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition, Edition 2*. Morgan Kaufmann, 2016.

[7] "Myth busted: General purpose cpus cant tackle deep neural network training," http://itpeernetwork.intel.com/myth-busted-general-purpose-cpus-cant-tackle-deep-neural-network-training/.

[8] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[9] T. Willhalm and N. Popovici, "Putting intel® threading building blocks to work," in *Proceedings of the 1st international workshop on Multicore software engineering*. ACM, 2008, pp. 3–4.

[10] J. Jeffers and J. Reinders, *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*. Morgan Kaufmann, 2015.

[11] "The intel(r) deep learning framework," https://github.com/01org/idlf.

[12] "Intel(r) math kernel library for deep neural networks," https://github.com/01org/mkl-dnn.

[13] Nervana, "Neon," https://github.com/NervanaSystems/neon.

[14] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2015, pp. 234–241.

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

[16] 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.

[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. Lee, A. Zlateski, A. Vishwanathan, and H. S. Seung, "Recursive training of 2d-3d convolutional networks for neuronal boundary detection," *arXiv preprint arXiv:1508.04843*, 2015.

[19] V. Jain, J. F. Murray, F. Roth, S. Turaga, V. Zhigulin, K. L. Briggman, M. N. Helmstaedter, W. Denk, and H. S. Seung, "Supervised learning of image restoration with convolutional networks," in *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*. IEEE, 2007, pp. 1–8.

[20] 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 ACM International Conference on Multimedia*. ACM, 2014, pp. 675–678.

[21] A. Viebke and S. Pllana, "The potential of the intel xeon phi for supervised deep learning," *arXiv preprint arXiv:1506.09067*, 2015.