ZNN - Fast and Scalable Algorithm for 3D ConvNets on Multi-Core and Many-Core Shared Memory Machines

Aleksandar Zlateski
Massachusetts Institute of Technology
Cambridge, MA
zlateski@mit.edu

Kisuk Lee
Massachusetts Institute of Technology
Cambridge, MA
kisuklee@mit.edu

H. Sebastian Seung
Princeton University
Princeton, NJ
sseung@princeton.edu

***Abstract*—Convolutional networks (ConvNets) have become a popular approach to computer vision. It is important to accelerate ConvNet training, which is computationally costly. We propose a novel parallel algorithm based on decomposition into a set of tasks, most of which are convolutions or FFTs. Applying Brent's theorem to the task dependency graph implies that linear speedup with the number of processors is attainable within the PRAM model of parallel computation, for wide network architectures. To attain such performance on real shared-memory machines, our algorithm computes convolutions converging on the same node of the network with temporal locality to reduce cache misses, and sums the convergent convolution outputs via an almost wait-free concurrent method to reduce time spent in critical sections. We implement the algorithm with a publicly available software package called ZNN. Benchmarking with multi-core CPUs shows that ZNN can attain speedup roughly equal to the number of physical cores, for machines with up to 40 cores. We also show that ZNN can attain over 90x speedup on a many-core CPU (Xeon Phi™ Knights Corner). These speedups are achieved for network architectures with widths that are in common use. The task parallelism of the ZNN algorithm is suited to CPUs, while the SIMD parallelism of previous algorithms is compatible with GPUs. Through examples, we show that ZNN can be either faster or slower than certain GPU implementations depending on specifics of the network architecture, filter sizes, and density and size of the output patch. ZNN may be less costly to develop and maintain, due to the relative ease of general-purpose CPU programming.**

TODO: Maybe add a sentence about how ZNN can work on arbitrary topology networks, bah

TODO: Don't like up to 40 cores (that's how much we have tested, ZNN will work for more cores)

I. Introduction

I will introduce myself!

A. Backpropagation

A standard formulation of supervised learning starts with a parametrized class of mappings, a training set of desired input-output pairs, and a loss function measuring deviation of actual output from desired output. The goal of learning is to minimize the average loss over the training set. A popular minimization method is stochastic gradient descent. For each input in sequence, the parameters of the mapping are updated in minus the direction of the gradient of the loss with respect to the parameters.

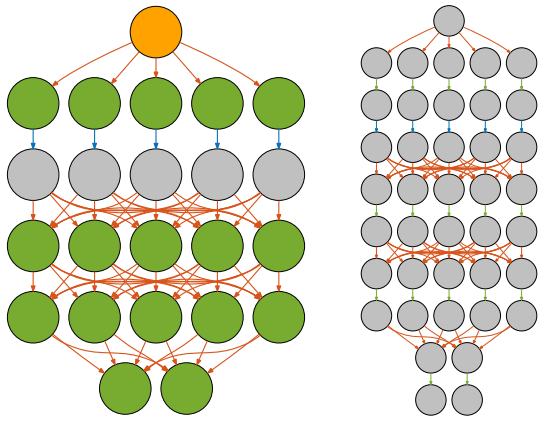


Fig. 1: Standard (a) representation of a ConvNet and ConvNet's computation graph (b). Red edges represent convolutions.

Here we are concerned with a class of mappings known as a convolutional network (ConvNet).

II. Computation graph

We define a ConvNet using a directed acyclic graph (DAG), called the *computation graph* (Fig. 1). Each node represents a 3D image, and each edge some filtering operation on a 3D image. (2D images are a special case in which one of the dimensions has size one.) The possible filtering operations are:

- Convolution
- Max-pooling
- Max-filtering
- Transfer function

If multiple edges converge on a node, the node sums the outputs of the filtering operations represented by the edges. For convenience, the discussion below will assume that images and kernels have isotropic dimensions, though this restriction is not necessary for ZNN.

Convolution A weighted linear combination of voxels within a sliding window is computed for each location of

Pass	Pooling	Filtering	Non-linearity
Forward	$f \cdot n^3$	$f \cdot 3n^3 \log k$	$f \cdot n^3$
Backward	$f \cdot n^3$	$f \cdot n^3$	$f \cdot n^3$
Update	–	–	$f \cdot n^3$

TABLE I: Complexities of pooling, filtering and non-linearity operations.

the window in the image. The set of weights of the linear combination is called the kernel. If the input image has size n^3 and the kernel has size k^3, then the output image has size $n'^3 = (n-k+1)^3$. Image size decreases because an output voxel only exists when the sliding window is fully contained in the input image. (This is known as a `valid` convolution in MATLAB.) The convolution is allowed to be sparse, meaning that only every sth image voxel within the sliding window enters the linear combination.

Max-pooling divides an image of size n^3 into blocks of size p^3, where n is divisible by p. The maximum value is computed for each block, yielding an image of size $(n/p)^3$.

Max-filtering The maximum within a sliding window is computed for each location of the window in the image. For a sliding window of size p^3, max-filtering yields an image of size $(n-p+1)^3$. The reduction in image size is the same as in convolution.

Transfer function adds a number called the bias to each voxel of the image and then applies a nonlinear function to the result. The nonlinear function is typically monotone nondecreasing. Common choices are the logistic function, the hyperbolic tangent and commonly used rectify linear function.

The computational complexities of max-pooling, max-filtering, and transfer function are shown in Table I. Convolution is more expensive, and its complexity will be discussed later.

Although ZNN works for a general computation graph, ConvNets in common use typically have the following properties:

- Convergent edges in the graph are convolutions; a sole incoming edge to a node is a nonlinear filtering operation.
- Nodes with convergent edges are not adjacent in the graph, but are separated from each other by nonlinear filtering edges. This is a reasonable constraint, because a composition of two convolutions can be collapsed into a single convolution, thereby simplifying the graph.
- The graph has a layered organization in which all edges in a layer represent operations of the same type.

A. Sliding window max-pooling ConvNet

A max-pooling ConvNet in the context of visual object recognition is a special case of the definition given above. Max-filtering is not used, and the input image is transformed into an output image consisting of a single pixel/voxel.

In a context where localization and detection are desired in addition to recognition, one can slide a window over a large image, and apply the max-pooling ConvNet at each location of the window [1]. However, it is computationally wasteful to literally implement the computation in this way. It turns out to be more efficient to implement a sliding window max-pooling ConvNet using a max-filtering ConvNet. Each max-filtering increases the sparsity of all subsequent convolutions by a factor equal to the size of the max-filtering window. This approach has been called *skip-kernels* [1] or *filter rarefaction* [2], and is equivalent in its results to *max-fragmentation-pooling* [3], [4].

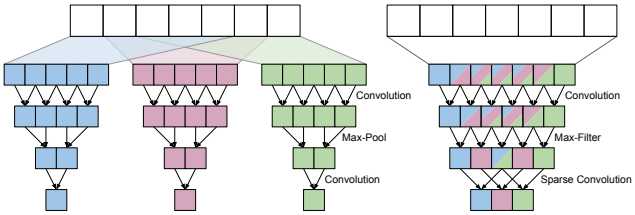


Fig. 2: Sliding window max-pooling ConvNet vs max-filtering ConvNet with sparse convolution

ZNN is more general, as the sparsity of convolution need not increase in lock step with max-filtering, but can be controlled independently.

Why is this useful? Patch training?

III. Backpropagation learning

The trainable parameters in a ConvNet are the kernels and biases. We will refer to these parameters as the weights, as they enter into weighted linear combinations. The backpropagation algorithm is a way of calculating the gradient of the loss function with respect to the weights. For each input, the calculation proceeds in several phases:

1) Obtain an input and desired output from the training set.
2) *Forward pass* - compute the actual output of the ConvNet from the input image.
3) Compute the gradient of the loss function with respect to the actual output.
4) *Backward pass* - Compute the gradient of the loss function with respect to the voxels of the output image at each node.
5) *Weight update* - Compute the gradient of the loss function with respect to the kernels and biases, and update these parameters in the direction of minus the gradient.

The forward pass has already been described above. ZNN implements several possibilities for the loss function, such as the Euclidean distance between the actual and desired outputs.

A. Backward pass

The starting point of the backward pass is the gradient of the loss function with respect to the voxels of the output node. The backward pass computes the gradient of the loss function with respect to the voxels of the images at the rest of the nodes. It turns out that the backward pass can be represented by another graph that looks the same as the forward computation graph, except that the direction of every edge is reversed.

Every edge in the backward computation graph is multiplication by the Jacobian matrix of the operation represented by the corresponding edge in the forward computation graph.

Convolution Convolution in the forward pass becomes convolution in the backward pass. The kernel is the same, except that it is reflected along all three dimensions. Reflecting an N-dimensional image along all dimensions is easily implemented through a one-dimensional flipping of the memory used by the image. Alternatively, one could store one bit of metadata (regular or flipped) about the state of the image.

If the input image has size n^3 and the kernel has size k^3, then the output image has size $n'^3 = (n + k - 1)^3$. Image size increases because an output voxel exists whenever the sliding window has some overlap with the input image. (This is known as a `full` convolution in MATLAB.)

Max-pooling Within each block, all voxels are zeroed out except for the one that was identified as the maximum within that block in the forward pass. An image of size n^3 is expanded into an image of size n^3p^3.

Max-filtering need description here

Transfer function Every voxel is multiplied by the derivative of the transfer function for the corresponding voxel in the forward pass.

B. Weight update

For a convolution going from node a to node b in the forward computation graph, the gradient of the loss with respect to the kernel is computed by convolving the reflected image at node a in the forward pass with the image at node b in the backward pass. A `valid` convolution is performed.

For a bias at node a, the gradient of the loss is calculated as the sum of all elements in the image at node a in the backward pass.

IV. Direct vs. FFT convolution

When the kernel sizes are large, the computational complexity of a single convolution can be reduced via FFT techniques [5], [6]. In the case of a layered network, the complexity of a fully connected convolutional layer with f input feature maps and f' output feature maps, for both direct and FFT-based convolution, are shown in the first two columns of Table II.[1]

Further optimization can be achieved by caching the FFTs of images and kernels obtained during the forward pass for reuse during the backward pass and weight update. In this case when computing the gradients we only need to calculate the FFT transform of the loss with respect to the output that is flipped in all dimensions, and reuse the value for both the backward and update phase.

The complexities of the three phases with FFT caching optimization are shown in the third column of the Table II. This method requires more memory but reduces computational complexity by approximately a third compared to the ones proposed in [5], [6].

[1] Note that our values differ from the ones in [5] as we have carefully examined the difference in complexity between `full` and `valid` convolutions

V. Task decomposition

The edges of the ConvNet's computation graph represent computation, whereas the nodes accumulate the results of their incident edges. For each edge type we introduce up to three different task types. All edge types will have a task for forward and backward pass computation. The edges with trainable parameters (convolutions and transfer functions) will also have a separate task for the update phase.

A. Task types

Forward The forward task of an edge $e = (u, v)$ is the application of some operation e.FORWARD-TRANSFORM to a 3D image accumulated in u, and addition of the resulting 3D image to the sum being accumulated by v.

Backward The backward task of an edge $e = (u, v)$ is the application of some operation e.BACKWARD-TRANSFORM to a 3D image accumulated in v, and addition of the resulting 3D image to the sum being accumulated by u.

Update An update task exists only for an edge $e = (u, v)$ with trainable parameters (kernel in the case of convolution or bias in the case of transfer function). The task alters the trainable parameters based on the images at u and v.

Training sample There are additional two training sample tasks. One task obtains a training sample used for a single round of training, and the other one calculates the gradient of the loss with respect to the network output.

B. Task dependency graph

A task can be executed when all the data required for its execution has been computed. The forward task of an edge $e = (u, v)$ depends on forward pass tasks of all edges $(w, u) \in E$. The backward task of the same edge depends on the backward task of all edges $(v, w) \in E$. Finally the update task of an edge depends on both forward and backward tasks of the same edge. Additionally, if there was a backward pass executed before the current forward pass, the forward pass task of e also depends on the previous update task of e.

Figure 3 shows the task dependency graph of the network shown in Figure 1. The bottom part of the graph contains tasks of a forward pass, whereas the top part contains the tasks of the previous backward pass. The topmost dark red circle nodes represent the tasks that calculate the gradient of the loss with respect to the output of the network obtained in the previous forward pass. The yellow circle in the middle represents the task generating the input sample for the current forward pass. Note that there are no update tasks for pooling/filtering. The task dependency graph represents tasks required to execute steps $3-5$ of the training procedure followed by steps 1 and 2 of the next round. (explain that this is better for analysis?)

C. Theoretically achievable speedup

We estimate the theoretically achievable speedup on different number of processors by analyzing layered networks where every convolutional layer is fully connected. Assuming one floating point instruction per time-step (cycle), we first estimate time required to perform one round of training on

Pass	Direct	FFT-based	FFT-based (Cached)
Forward	$f' \cdot f \cdot n'^3 \cdot k^3$	$3Cn^3 \log n[f' + f + f' \cdot f] + 4f' \cdot f \cdot n^3$	$3Cn^3 \log n[f' + f + f' \cdot f] + 4f' \cdot f \cdot n^3$
Backward	$f' \cdot f \cdot n'^3 \cdot k^3$	$3Cn^3 \log n[f' + f + f' \cdot f] + 4f' \cdot f \cdot n^3$	$3Cn^3 \log n[f' + f] + 4f' \cdot f \cdot n^3$
Update	$f' \cdot f \cdot n'^3 \cdot k^3$	$3Cn^3 \log n[f' + f + f' \cdot f] + 4f' \cdot f \cdot n^3$	$3Cn^3 \log n[f' \cdot f] + 4f' \cdot f \cdot n^3$
Total	$3f' \cdot f \cdot n'^3 \cdot k^3$	$9Cn^3 \log n[f' + f + f' \cdot f] + 12f' \cdot f \cdot n^3$	$6Cn^3 \log n[f' + f + f' \cdot f] + 12f' \cdot f \cdot n^3$

TABLE II: Computational complexity of a fully connected convolutional layer

Edge type	COMPUTE-FORWARD	COMPUTE-BACKWARD	COMPUTE-UPDATE
Convolution	$x^{out} = x^{in} *_{\mathrm{V}} w$	$\frac{\partial L}{\partial x^{in}} = \frac{\partial L}{\partial x^{out}} *_{\mathrm{F}} w^F$	$\frac{\partial L}{\partial w} = (x^{in})^F *_{\mathrm{V}} \frac{\partial L}{\partial x^{out}}$
Transfer function	$x^{out} = \sigma(x^{in} + b)$	$\frac{\partial L}{\partial x^{in}} = \frac{\partial L}{\partial x^{out}} \frac{\partial \sigma}{\partial x^{in}}$	$\frac{\partial L}{\partial b} = \sum_{i,j,k} \frac{\partial L}{\partial x^{out}_{(i,j,k)}}$
Max/Min Pooling	$x^{out}_{(i,j,k)} = \min_{(p,q,r) \in W} x^{in}_{(p,q,r)}$	$6Cn^3 \log n + 4n^3$	$3Cn^3 \log n + 4n^3$
Filtering	$3n'^3 \cdot k^3 + n'^3 \lceil \log_2 f \rceil + n^3 \lceil \log_2 f' \rceil$	$18Cn^3 \log n + 4n^3[1 + \lceil \log_2 f \rceil + \lceil \log_2 f' \rceil]$	$12Cn^3 \log n + 4n^3[1 + \lceil \log_2 f \rceil + \lceil \log_2 f' \rceil]$

TABLE III: Caption.

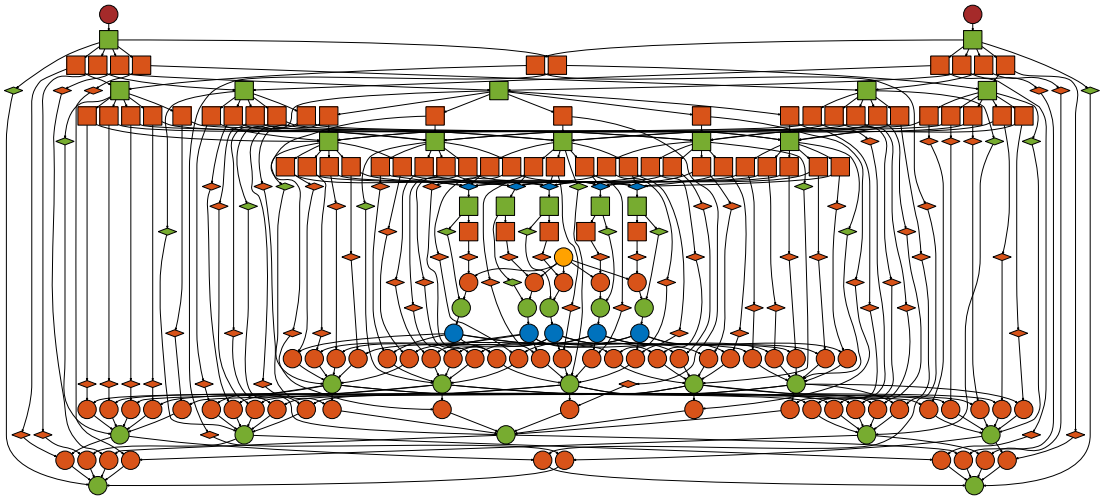


Fig. 3: Task dependency graph of the ConvNet in Figure 1
The tasks are color/shape coded. The square shape represents a backward type task, a circle represents a forward type task, and a diamond represents an update task. The green color represents a non-linearity task (transfer function), the red color represents convolution task and blue represents pooling/filtering task.

an infinite number of processors available, by estimating the time required for the forward and backward passes as well as the update phase for each layer type of the network. For the analysis we use the same notation as in the previous section.

In the fully connected convolutional layer, when using direct convolution, all $f \cdot f'$ convolutions can be done at the same time and take $n'^3 \cdot k^3$ time-steps. Each of f' output nodes then accumulate results of f convolutions which can be done in $n^3 \lceil \log_2 f \rceil$ time per output node[2]; and all the output nodes can be done in parallel, giving us total of $n'^3 \cdot k^3 + n'^3 \lceil \log_2 f \rceil$. With analogous analysis we get that the time required for the backward pass of the layer equals to $n'^3 \cdot k^3 + n^3 \lceil \log_2 f' \rceil$. During the update phase, all the updates can be done in parallel and require total of $n'^3 \cdot k^3$ time-steps. In the case of FFT-based convolution, the FFTs of the input feature maps and all the filters can be done at the same time in $3Cn^3 \log n$ time-steps. The FFT of the output feature map is then obtained by accumulating point-wise products of the FFT of the input feature map with the FFT of the appropriate filter, taking $4n^3 \lceil \log_2 f \rceil$ time-steps. We can then compute the inverse FFT of all the output feature maps in $3Cn^3 \log n$ time-steps. The analysis for the backward pass is equivalent, requiring exact amount of time to perform as the forward pass. For the update phase we can compute the FFTs of the input feature maps and the gradients of the loss with respect to the outputs at the same time in $3Cn^3 \log n$ time-steps. In the case we have cached these values, no computation is necessary. For each edge we can compute the point-wise product of the two at the same time in $4n^3$, and finally compute the inverse transforms of the results in another $3Cn^3 \log n$ time-steps.

The total time required for the first convolutional layer in the network then equals to the sum of the time required for the forward and the backward pass plus the time required for the update phase. The total time required for all other convolutional layers equals to the sum of the time required for the forward and backward pass, as all the update computation can be done at the same time as the update computation is done for the first layer. All times required for a fully convolutional layer are shown in Table IV.

[2]Using the algorithm described in [7]

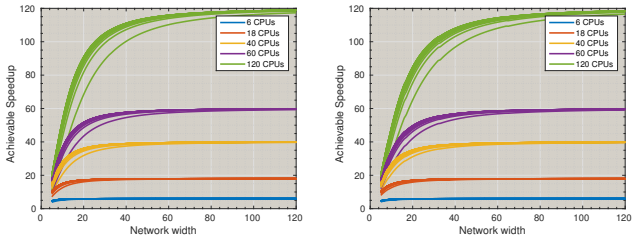


Fig. 4: Achieavle speedup of a ConvNet using (a) direct convolution (b) FFT-based convolution with caching enabled.

Applying the transfer function in each of f nodes in a layer can be done at the same time, as well as computing the gradient of the loss with respect to the input and the bias. This gives us total of n^3 time-steps required for computing the forward pass, backward pass and the update phase for the whole layer. Again the total time required by the first layer in the network is $3n^3$, and the time for all other layers is $2n^3$ as the update phase can be done at the same time as the update is being computed for the first such layer.

The pooling/filtering layers have the same number of input feature maps and output feature maps. All pulling/filtering operations can be done at the same time for both forward and backward pass. The Table V shows time-steps required for transfer functions and pooling/filtering layers.

Summing up the times required for each layer of a network from Tables II and I gives us T_1 - the time required for performing a single round of training in serial fashion (on a single processor); summing up the times from Tables IV and V gives us T_∞ - the time required to compute one round of training on an infinite amount of processors available.

We can estimate theoretically achievable speedup on P available processors by using Brent's theorem [7].

$$T_P \leq T_\infty + \frac{T_1 - T_\infty}{P} \tag{1}$$

Let S_P be speedup achieved using P processors, $S_p = \frac{T_1}{T_P}$.

$$S_P \geq \frac{S_\infty}{1 + \frac{S_\infty - 1}{P}} \tag{2}$$

As the computation is dominated by the fully connected convolutional layers, we can roughly estimate $\frac{T_1}{T_\infty} \approx \mathcal{O}(f^2)$. We can saturate P processors with networks whose width is approximately equal to $\sqrt{P}$ which means that we can get high utilization for networks with very modest widths. We show the plot of exactly computed theoretically achievable speedup for networks of different width and height in Figure 4.

VI. Task scheduling

Assuming no synchronization and communication overhead, Brent's theorem gives us the lower bound on the time required to perform a single pass of network training. We see that achievable speedup depends mostly on network width, and less on network depth, and that a high degree of utilization is achievable.

Pass	Pooling	Filtering	Non-linearity
Forward	n^3	$3n^3 \log k$	n^3
Backward	n^3	n^3	n^3
Update	–	–	n^3
Total First	–	–	$3n^3$
Total Other	–	–	$2n^3$

TABLE V: Time required to perform different operations on a full layer with infinite number of processors available.

In order to achieve the predicted performance we design our algorithm such that we minimize any synchronization overhead and increase temporal locality of computation in order to reduce cache misses.

The central quantity in our algorithm is a global priority queue that contains tasks that are ready to be executed together with their priority. A predetermined number of workers will then execute the tasks from the global queue.

A. Priority queue

Tasks are placed on a global queue when all non-update dependencies are satisfied. The only tasks with update task as requirements are forward tasks. The rationale behind this design choice is that if a forward task is scheduled for execution without the required update task being done, we will force execution of the update task followed by the forward task that requires the result of the update, hence increase the memory locality.

The tasks on the queue are sorted by priority. We chose the priorities in such a way to increase the temporal locality of the computation and minimize the latency of the computation. We introduce two unique strict orderings of the nodes in the ConvNet's computation graph based on the longest distance, in decreasing order, to any output and input node respectively. Nodes with the same distance will be ordered in some unique way. The priority of the forward task of an edge $e = (u, v)$ will be equal to position of the output node v in the ordering based on the distance to the output nodes, and similarly the priority of the backward task will equal to the ordering of u based on the distance to the input nodes. This ensures that we prioritize tasks with the longest path to a sink node in the task dependency graph and hence minimize latency. The strict ordering of the tasks with the same distance increases temporal locality by assuring that when multiple tasks with the same distance are scheduled we prefer to execute ones computing 3D images that have to be accumulated in the same sum, thus increasing the probabiltiy of the memory accessed being in the cache.

The update tasks will have the lowest priority of all tasks. Their execution will be forced when their result is required for the forward pass, which increases cache locality as the result will be used immediately. The only other time the update tasks will be executed is if there's no other forward or backward tasks ready to be executed.

Pass	Direct	FFT-based	FFT-based (Cached)
Forward	$n'^3 \cdot k^3 + n'^3\lceil \log_2 f \rceil$	$6Cn^3 \log n + 4n^3\lceil \log_2 f \rceil$	$6Cn^3 \log n + 4n^3\lceil \log_2 f \rceil$
Backward	$n'^3 \cdot k^3 + n^3\lceil \log_2 f' \rceil$	$6Cn^3 \log n + 4n^3\lceil \log_2 f' \rceil$	$6Cn^3 \log n + 4n^3\lceil \log_2 f' \rceil$
Update	$n'^3 \cdot k^3$	$6Cn^3 \log n + 4n^3$	$3Cn^3 \log n + 4n^3$
Total First	$3n'^3 \cdot k^3 + n'^3\lceil \log_2 f \rceil + n^3\lceil \log_2 f' \rceil$	$18Cn^3 \log n + 4n^3[1 + \lceil \log_2 f \rceil + \lceil \log_2 f' \rceil]$	$12Cn^3 \log n + 4n^3[1 + \lceil \log_2 f \rceil + \lceil \log_2 f' \rceil]$
Total Other	$2n'^3 \cdot k^3 + n'^3\lceil \log_2 f \rceil + n^3\lceil \log_2 f' \rceil$	$12Cn^3 \log n + 4n^3[\lceil \log_2 f \rceil + \lceil \log_2 f' \rceil]$	$12Cn^3 \log n + 4n^3[\lceil \log_2 f \rceil + \lceil \log_2 f' \rceil]$

TABLE IV: Time required to perform different operations on fully connected convolutional layers with infinite number of processors available.

B. Worker loop

The tasks are executed by N workers, where N is experimentally determined, and as shown below, in most cases N should equal the number of virtual cores available. Each worker picks up and executes a task with the highest priority on the queue.

Forward tasks When forward tasks are scheduled for execution the worker first checks the state of the required update task, which can be one of the following:

1) **Pending**, in which case the update task is in the queue waiting to be executed. In this case the worker removes the task from the queue, executes the update task followed by the scheduled forward task.
2) **Completed**, the worker then just executes the forward task pass.
3) **Executing**, in this case, some other worker is currently executing the update task. The current worker *attaches* the scheduled forward task to be executed by the worker currently executing the update task upon completion of the update task. The current worker then just picks up the next task from the queue.

In the first two cases, upon the completion of executing the forward task, the worker will check whether there are more tasks with all-but update tasks satisfied, and will insert them to the queue.

Algorithm 1 Forward Task algorithm

```
DO-FORWARD(e = (u, v), x)
1  x^out = e.FORWARD-TRANSFORM(x)
2  if ADD-TO-SUM(v.sum, x^out)
3      y = GET-SUM(v.sum)
4      for e' ∈ v.out-edges
5          t = MAKE-FORWARD-TASK(e', y)
6          ENQUEUE(e'.fwd_priority, t)

EXECUTE-FORWARD-TASK(e, x)
1  t = TASK(DOFORWARD(e, x))
2  FORCE(e.update_task, t)
```

Backward tasks The worker just executes the backward task and inserts the update task to the queue if necessary.

Update tasks The worker executes the task, and upon completion checks whether some other worker had *attached* a forward task to be executed. If there is an attached forward task, the worker executes it and checks whether there are more tasks to be placed on the queue.

Algorithm 2 Backward task algorithm.

```
EXECUTE-BACKWARD(e = (u, v), dLdF)
1  dLdX = e.COMPUTE-GRADIENT(dLdF)
2  if e.trainable
3      e.update-task = TASK(UPDATE(e, dLdF))
4      ENQUEUE(lowest_priority, e.update_task)
5  v = e.v^out
6  if v.add_backward(dLdX)
7      for e' ∈ v.in_edges
8          y = v.get_backward_sum()
9          t = TASK(BACKWARD(e', y))
10         ENQUEUE(e'.bwd_priority, t)
```

Algorithm 3 Update task algorithm.

```
UPDATE(e, dLdF)
1  dLdB = e...
2  e.b = ADVANCE(e, dLdB)
```

C. Synchronization issues

Minimize critical sections, the code that can be executed by a single thread at the time. The main three points in the algorithm that require synchronization are memory management (allocation/deallocation), operations on the global task queue and concurrent summations.

1) Queue operations: The operations on the global task priority queue have to be synchronized. The global queue is implemented as a heap of lists lowering the complexity of insertion and deletion from $\log N$ to $\log K$, where N is the total number of tasks in the queue and K is the number of distinct values for the priority of the tasks inside the queue. Depending on the network structure, this number can be much smaller than the total number of tasks in the queue.

2) Wait-free concurrent summation: Multiple tasks that are being executed in parallel might need to add their result to the same accumulated sum. The additions to the sum have to be synchronized - only one thread/task is allowed to change the sum. A simple strategy to wait until all other threads have finished updating the sum can yield large performance degradation as summing large 3D images can be a time consuming

operation. To minimize the synchronization time we decide to only manipulate the pointer to the currently accumulated sum. The method CONCURRENT-SUM in Algorithm 4 describes the procedure of adding values of a 3D image pointed to by v to an accumulated sum. The idea is to grab the pointer to the sum inside the critical section (lines 5-11). If the pointer equals NIL we set it to the pointer provided that contains the values to be added to the sum. Otherwise we set the pointer to the sum to NIL and then add the values of the accumulated sum to the values of the image to be added to the sum outside the critical section. We then repeat trying to add the increased values to the sum in the same fashion.

Algorithm 4 Wait-free concurrent summation algorithm

```
ADD-TO-SUM(S, v)
 1  v' = NIL
 2  last = FALSE
 3  repeat
 4      ACQUIRE(S.lock)
 5          if S.sum == NIL
 6              S.sum = v
 7              v = NIL
 8              S.total = S.total + 1
 9              last = (S.total == S.required)
10          else v' = S.sum
11              S.sum = NIL
12      RELEASE(S.lock)
13      if v == NIL
14          return last
15      else ADD-TO(v, v') // v = v + v'
16  until

GET-SUM(S)
1  v' = S.sum
2  S.sum = NIL
3  S.total = 0
4  return v'
```

3) Memory management: In order to minimize the synchronization time during memory allocation and de-allocation ZNN implements two custom memory allocators, trading speed for some memory usage overhead. One is dedicated to 3D images, which are usually large, and the other one is dedicated to small objects used in auxiliary data structures. Both allocators maintain 32 global pools of memory chunks. Each pool i, $i \in 0 \dots 31$ contains chunks of sizes of 2^i. Lock-free queues, as described in [8] and implemented as a part of the boost [9] library are used to implement the pool operations. The only difference between the allocators is the memory alignment - the 3D image memory allocator ensures proper memory alignment for utilizing SIMD instructions. No memory is shared between the two allocators.

When a chunk of memory of size s is requested, first s is rounded up to the nearest power of 2. The appropriate pool is examined for available memory chunks. If there's an available chunk we return it and remove it from the pool. If no chunks are available we allocate one from the system and return it.

When de-allocating a chunk memory, it is simply added to the appropriate pool, and no memory is ever returned to the system. This means that the memory usage of our program can never decrease. In practice, as the ConvNet training consist of a single loop performing the same work, our memory usage peaks after a few rounds.

In the worst case this strategy can lead to $2\times$ memory usage overhead; however the available memory to the CPU is rarely a limiting factor in training a network. In future, we might consider implementing more advanced memory allocators, such as ones with thread-local pools in addition to the global pool, or ones with higher granularity of available chunk sizes to reduce the size overhead.

Using **jemalloc** [10], [11] for small object allocation yields similar performance with a bit less overhead, however we decide to use our own implementation as a default one, in order to minimize the number of dependencies and make the code more portable.

CPU	Frequency	Cores/Threads
Intel® Xeon™ E5-2666 v3	2.9 GHz	8 cores/16 threads
Intel® Xeon™ E5-2666 v3	2.9 GHz	18 cores/36 threads
Intel® Xeon™ E7-4850	2.0 GHz	40 cores/80 threads [4]
Intel®Xeon Phi™5110P	1.053 GHz	60 cores/240 threads [5]

TABLE VI: Machines used for the experiments

VII. MEASUREMENTS

To evaluate the efficiency and usability of the proposed algorithm, we pursue the answers to the following two questions:

1) What is the scalability of the parallel algorithm - how does the speed scale with depending on the number of available processors.
2) How does ZNN's performance compare to the current state of the art implementations of ConvNets.

To answer the first question we benchmark the speed of training networks with different structures on different multi-core and many-core systems and compare the achieved speedup to the serial algorithm. We used 8 and 18 core readily available machines on Amazon Web Services (AWS) as well as a 40 core 4-way CPU system and a Xeon Phi™Knights Corner. We have used Intel compiler (version 15.0.2) with Intel MKL (version 11.2) libraries for FFTs and direct convolution. The list of machines used is shown in Table VI[3].

We obtain part of the answer to the second question by comparing ZNN's performance to the publicly available implementation of ConvNets. ZNN is the first publicly available implementation optimized for CPUs, and that supports 3D + sliding window output... yada yada.... we add this later when we figure out what we will compare to.

[3]The 8 and 18 core machines correspond to AWS instances c4.4xlarge and c4.8xlarge

We also mention that ZNN has less limitations so we can only compare a subset of ZNNs abilities to the ones currently available. Mention that ZNN can do arbitrary graphs, sliding window, multiscale, etc...

A. Scalability

To measure the scalability of our algorithm we have benchmarked 2D and 3D ConvNets of different widths. The 2D networks are implemented by setting one of the dimensions to be one. They had 8 layers, with layers 2 and 4 being pooling layers with a window of size 2×2 and other layers being convolutional layers with the kernel size of 11×11. We measure the speed of patch training evaluated on the sliding window with the output size being 48×48. The 2D networks had only 6 layers with two pooling layers (2 and 4) and 3 convolutional layers with the filter size of $3 \times 3 \times 3$. The 2D networks use FFT based convolution whereas the 3D networks use direct convolution. We chose to present these networks in order to cover wider range of possible settings for the network; however using deeper 3D networks or changing the FFT based convolution with direct or vice versa, as well as changing the filter sizes yields very similar results.

We have used different numbers of worker threads for networks of different width. Figure 5 shows the obtained results. Different lines correspond to networks of different width. For the desktop CPUs we observe the linear increase in speed up to the number of cores, then linear increase with a lower slope up to the number of virtual threads. As predicted by Brent's law, the networks have to be wide enough in order to achieve high efficiency. The peak efficiency is reached for networks of width 20 on all desktop processors. The results for Xeon Phi™ show linear increase until the number of virtual cores then linear increase with a lower slope until double that number. For networks wide enough the speed still increases up to the number of hardware threads [6].

The maximal achieved speedups for a networks of different widths are shown in Figure 7.

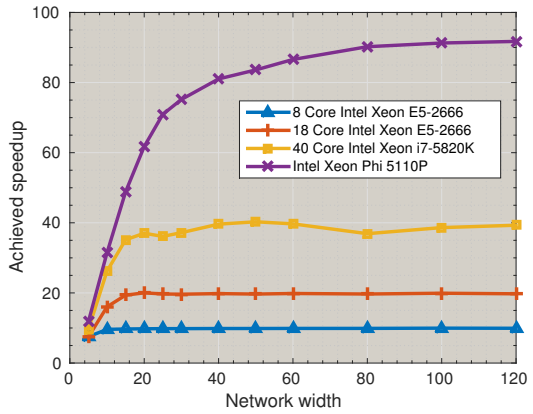


Fig. 7: Achieved speedups on a 2D network compared to the serial algorithm. Absolute speeds can be obtained by multiplying the speedup with the speeds of the serial algorithm which are X,Y,Z,W updates/sec for 8, 18 and 40 core Xeon CPUs and Xeon Phi respectively.

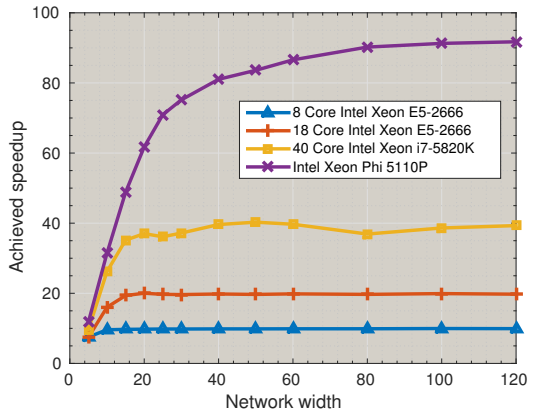


Fig. 8: Achieved speedups on a 3D network compared to the serial algorithm.

B. GPU comparison

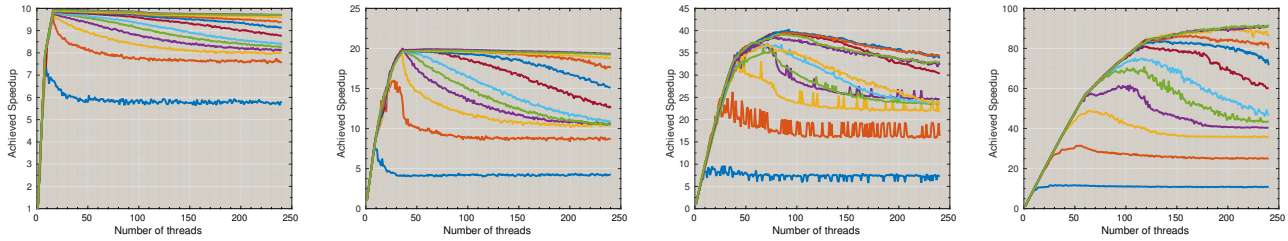
In order to compare the ZNN's performance versus the state of the art GPU implementation we benchmark modern pooling networks. We decide on a modest network width of 40 and vary the sizes of the filters as well as the size of the output patch. As there is no publicly available GPU implementation that can work with dense output, all our comparisons are done for sparse output patch. ZNN is executed on an 18 core AWS machine and the GPU implementations are using the Titan X.

For the 2D networks we benchmark ZNN against caffe [12], both the default implementation and an implementation using cudnn [13] as well as the default implementation of theano [14] implementations. For 3D we only use theano, as the official release of caffe still doesn't support 3D ConvNets. We have carefully optimized the speed measuring code for the GPU implementations, and made it publicly available in the ZNN's repository.

The comparison of 2D ConvNets is shown in Figure 9 and on 3D ConvNets in Figure 10. The relative performances of ZNN increases with the increasing output patch size. For the 2D networks and filter sizes of 20×20 ZNN performances are comparable to the ones of Caffe using cuDNN. For filter sizes of 30×30 ZNN already performs slightly better both Theano and Caffe. And for the filter sizes of 40×40 ZNN performs two times faster than the next fastest implementation. Note that some data is missing for Caffe; in those cases Caffe was not able to handle the network of given size. For 3D ConvNets ZNN achieves comparable performance even for modest filter sizes of $5 \times 5 \times 5$ and outperforms Theano for filter sizes of $7 \times 7 \times 7$.

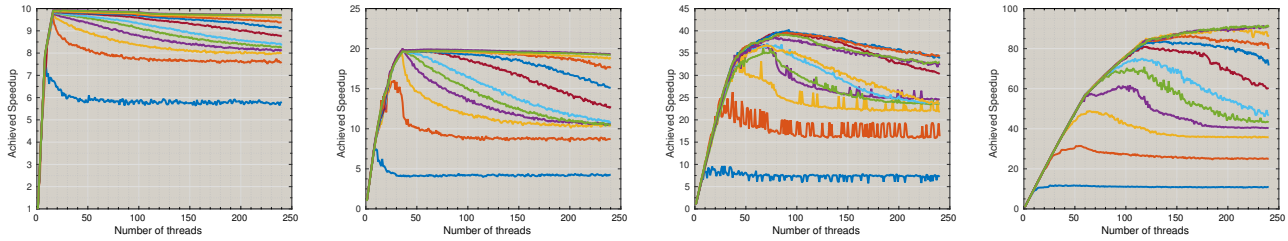
ZNN might not seem very competative for 2D networks as very large filters are not that common. However, until recently

[6] Xeon Phi™ has hardware threads which differ from virtual thread technology of the desktop Xeon™ processors



(a) (b) (c) (d)

Fig. 5: 2D networks scalability. Bottom-up the different lines correspond to networks of widths $5, 10, 15, 20, 25, 30, 40, 50, 60, 80, 100, 120$



(a) (b) (c) (d)

Fig. 6: 3D network scalability

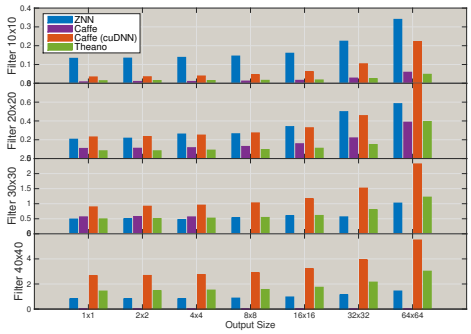


Fig. 9: Comparison of ZNN, Caffe and Theano for 2D ConvNets.

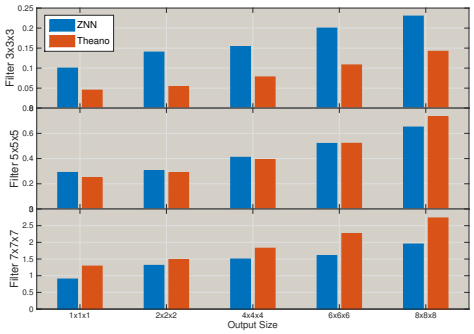


Fig. 10: Comparison of ZNN and Theano for 3D ConvNets.

training networks with such large filters was impratical. Using networks with large filters might became more common in future, and such networks can be efficiently trained with ZNN. On the other, for 3D ConvNets ZNN is very competitive even for modest sizes and commonly used filter sizes. Reference Nature paper with 7x7x7 filter sizes?

Aside from speed the other limiting factor in training deep ConvNets Theano was not able to train networks with larger filter sizes. The memory usage of network with filter sizes of $7 \times 7 \times 7$ ranged from 5.8 to 11.3 GB depending on the output patch size. Titan X has only 12 GB of RAM available which is usually much less than the memory available to the CPU required by ZNN, and until recently GPUs had even less memory. ZNN, hence, allows for training of much larger networks.

Even though ZNN is not meant to be direct competitior to the GPU implementation, but rather a more flexible framework allowing for (more stuff like multi scale dense output, etc..) we see that ZNN has comparable performance on commonly used networks of modest sizes.

VIII. Results

This is where we maybe add results obtained by using ZNN. Maybe try to use multiscale functionality? Not sure what is the best approach here - kisuk?

A. Implementation Details

ZNN is implemented in C++ and is publicly available under XXX license https://github.com/zlateski/znn-release Uses fftw or intel MKL for FFT and either provided naive code or intel MKL for direct convolution.

The modular design of ZNN allows for easy replacement of different parts of the algorithm. ZNN also provides different scheduling strategies such as simple FIFO or LIFO as well as more complex ones based on work stealing [15]. The alternative scheduling strategies perform 10-20% worse than the one proposed in the paper for most networks. However some very specific networks can benefit from alternative scheduling algorithms. Future work can include automatic detection of the best scheduling strategy.

IX. Related work

GPU [12], [16], [14]. FFT [5], [6]. Sliding window [4], [3], [1]. Distributed networks [17].

One attempt [18] to improve neural nets on Xeon Phi™.

X. Conclusions

Draft below:

ZNN's design allows for easy addition of new types of layers. Adding additional features to the existent GPU frameworks requires tedious SIMD programming. GPU implementations are mostly improve the performances on the specific types of networks, and limit the users to batch training.

ZNN focuses on enabling research of new types of networks. The generic programming model and task based parallelism allow for the users to easily add new features by providing a serial algorithm, and ZNN will take care of efficiently training the network on a multi-core machine.

To our knowledge ZNN is the first publicly available ConvNet package to support sliding window and multiscale networks.

References

[1] 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.

[2] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," *arXiv preprint arXiv:1411.4038*, 2014.

[3] A. Giusti, D. C. Cireşan, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Fast image scanning with deep max-pooling convolutional neural networks," *arXiv preprint arXiv:1302.1700*, 2013.

[4] J. Masci, A. Giusti, D. Ciresan, G. Fricout, and J. Schmidhuber, "A fast learning algorithm for image segmentation with max-pooling convolutional networks," in *Image Processing (ICIP), 2013 20th IEEE International Conference on*, pp. 2713–2717, IEEE, 2013.

[5] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," in *International Conference on Learning Representations (ICLR2014)*, CBLS, April 2014.

[6] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A gpu performance evaluation," *arXiv preprint arXiv:1412.7580*, 2014.

[7] J. Gustafson, "Brents theorem," in *Encyclopedia of Parallel Computing* (D. Padua, ed.), pp. 182–185, Springer US, 2011.

[8] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pp. 267–275, ACM, 1996.

[9] T. Blechmann, "Boost lockfree library." http://www.boost.org/libs/lockfree/, 2008.

[10] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.

[11] J. Evans, "Scalable memory allocation using jemalloc." https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919, 2011.

[12] 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*, pp. 675–678, ACM, 2014.

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

[14] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a cpu and gpu math expression compiler," in *Proceedings of the Python for scientific computing conference (SciPy)*, vol. 4, p. 3, Austin, TX, 2010.

[15] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

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

[17] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012.

[18] L. Jin, Z. Wang, R. Gu, C. Yuan, and Y. Huang, "Training large scale deep neural networks on the intel xeon phi many-core coprocessor," in *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IPDPSW '14, (Washington, DC, USA), pp. 1622–1630, IEEE Computer Society, 2014.