PZnet: Efficient CPU ConvNet Inference Engine for 3D Medical Image Processing

Sergiy Popovych
Computer Science Dept.
Princeton University
popovych@princeton.edu

Davit Buniatyan
Computer Science Dept.
Princeton University
davit@princeton.edu

Alexander Zlateski
Computer Science Dept.
MIT
zlateski@csail.mit.edu

H. Sebastian Seung
Princeton Neuroscience Institute
Princeton University
sseung@princeton.edu

Kai Li
Computer Science Dept.
Princeton University
ckyli@princeton.edu

Abstract

Convolutional nets have been shown to achieve state-of-the-art accuracy in many biomedical image analysis tasks. To deploy convolutional nets in practical working systems, it is also important to solve the efficient inference problem. Namely, one should be able to apply an already-trained convolutional network to many large images using limited computational resources. 3D images are especially relevant because biological tissues are 3D, and data volumes are typically high for 3D. While it is common to use GPUs for convolutional net inference, there may be environments where CPUs are more abundant or accessible. In this paper we present PZnet, a CPU-only engine that can be used to perform inference for a variety of 3D convolutional net architectures. PZNet outperforms MKL-based CPU implementations of PyTorch and Tensorflow by more than 3.5x for the popular 3D U-net architecture. Moreover, based on current pricing of preemptible or spot instances, cloud CPU inference with PZnet is competitive in cost with cloud GPU inference, for U-net style architectures.

1 Introduction

Modern deep learning frameworks such as Theano [2], Caffe [4], Tensorflow [1] and Pytorch [3] are mostly optimized for processing of 2D images, and achieve lower hardware utilization on both CPU and GPU platforms for 3D tasks. In this work we show that CPU efficiency for 3D ConvNet inference can be improved by up to 8x, which results in higher utility of existing CPU infrastructure and makes CPU inference a competitive choice in the cloud setting.

The main contribution of this work is an inference-only deep learning engine called PZnet, which is specifically optimized for 3D inference on Intel Xeon CPUs. PZnet utilizes ZnnPhi[6], a state-of-the-art direct 3D convolution implementation. ZnnPhi relies heavily on template-based metaprogramming and requires a custom data layout, which makes it difficult to integrate into mainstream deep learning frameworks. For this reason, we created a special inference-only framework PZnet. A convolutional net can be trained using a mainstream deep learning framework, and then imported to PZnet when large-scale inference is required.

PZnet also provides a number of ways of fusing multiple layers into one, thereby reducing the amount of computation required during inference. These layer fusions are applicable to a number of convolutional net architectures, and can reduce inference time by up to 20%.

PZnet outperforms MKL-based CPU implementations of PyTorch and Tensorflow by 3-8x, depending on the network architecture and hardware platform. Moreover, we show that based on current cloud compute prices, PZnet CPU inference is competitive with cuDNN based GPU inference. For inference of a real-world residual 3D Unet architecture [5], PZnet is able to outperform GPU inference in terms of cost efficiency by over 50%. To the best of our knowledge, this is the first work to show CPU inference to beat GPU inference in terms of cloud cost.

2 PZnet

PZnet consists of 2 parts – a network generator and an inference API. Network generator compiles the provided model specifications into so-called *network files*. Network files are shared library objects that are distributed to worker machines. Workers run inference by accessing the models within the network files through PZnet python inference API.

PZNet employs ZnnPhi, which, to the best of our knowledge, is the most efficient 3D direct convolution implementation known up to date. ZnnPhi achieves high performance though utilizing SIMD instructions in a cache efficient way, and is compatible with SSE4, AVX, AVX2 and AVX512 SIMD instruction families.

ZnnPhi requires image and kernel data to conform to a specific data layout. The data layout stems from the way in which ZnnPhi utilizes SIMD instructions, and it prevents ZnnPhi from being directly pluggable into mainstream deep learning frameworks.

Additionally, ZnnPhi heavily relies on metaprogramming through C++ templates, which means that layer parameters, such as image and kernel sizes, have to be known during compile time. This allows ZnnPhi to rely on compile time optimizations in order to produce maximally efficient code for each parameter configuration. However, this adds another obstacle to integrating ZnnPhi into a mainstream deep learning framework. Most deep learning frameworks allow user-supplied C++ layer implementations, but they either require them as a compiled shared object or as generic source code. Since ZnnPhi layers need to be recompiled for each layer configuration, integration with mainstream deep learning frameworks cannot be achieved without implementing JIT compilation infrastructure.

In order to support ZnnPhi, PZnet compiler generates C++ source code which directly corresponds to the provided model. Then, Intel C++ Compiler is invoked in order to produce optimized shared library object files. All PZnet layers support ZnnPhi blocked memory layout. PZnet implicitly performs memory layout transformations for the input and output data tensors in order to provide standard input and output formats.

3 Optimizations

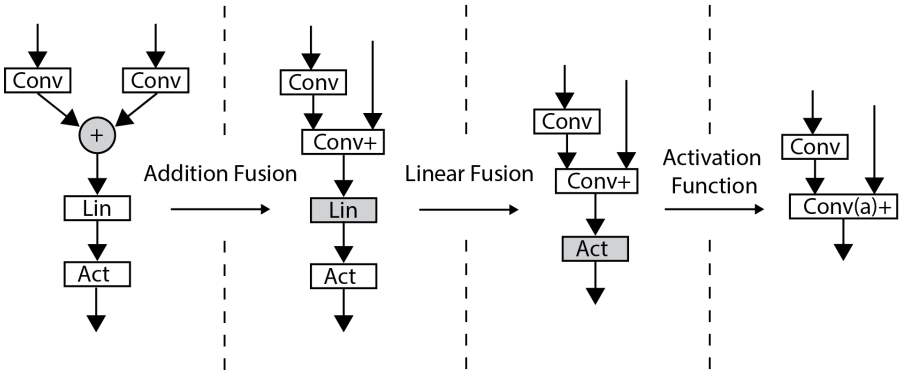

Figure 1: PZnet optimization flow

Optimizations performed by PZnet are mainly aimed at reducing the number of memory traversals introduced by miscellaneous layers (batchnorm, scale, activation, etc). We modify ZnnPhi primitives in order to perform operations required by several layers in one memory pass. The overall optimization flow of PZnet is as follows. First, we fuse convolutions with element-wise addition layers, which are

commonly introduced by residual connections. After element-wise layers are fused, more convolution layers immediately precede batch normalization and scaling layers. Both batch normalization and scale perform linear transformations of tensors during inference time. Since weights of the convolution layers can be modified in order to take account for subsequent linear transformation layers, we are able to fuse batch normalization and scales into convolution. After linear transformation layers are fused, convolutions are commonly followed by activation layers. We modify ZnnPhi primitives in order to apply activation function to the convolution outputs before they are written out to memory, thus fusing activation and convolution layers. Finally, after element-wise addition, linear transformation and activation layers are removed, most of convolution layer outputs are used inputs of the subsequent convolution layers. Thus, we can eliminate explicit input padding of the inputs by making convolution layers produce padded outputs, which saves additional memory traversals. Overall, optimization by $3.5 - 20\%$ is performed, depending on CPU parameters and network architecture.

4 Evaluation

The experiments are performed on major types of CPU and GPU compute instances from Amazon Web Services (AWS) and Google Cloud: AWS C4, AWS C5, Google Cloud Hasswell, Google Cloud Skylake. 3 versions of 3D Unet architecture are evaluated: original, symmetric and residual.

Table 1 compares CPU performance of PZnet and Tensorflow. Tensorflow version used for evaluation was compiled with MKL, FMA and AVX2 support. The results show that PZnet outperforms Tensorflow by more than 3.4x for all of the experiment settings.

Table 1: PZnet vs Tensorflow CPU performance

		Time per patch (sec)							
		PZnet				Tensorflow			
Network	Act	c4	c5	g Has	g Sky	c4	c5	g Has	g Sky
Original	ReLU	1.82	1.47	5.81	3.91	6.70	5.91	22.46	21.40
Original	ELU	1.89	1.55	5.99	4.07	6.45	5.64	22.89	22.60
Symmetric	ReLU	3.97	2.68	12.93	8.35	30.93	28.43	114.34	114.78
Symmetric	ELU	4.11	2.78	13.30	8.68	30.79	27.64	114.50	113.89
Residual	ReLU	1.61	1.48	5.00	4.62	8.96	7.95	22.75	22.97
Residual	ELU	1.67	1.52	5.20	4.84	8.26	7.40	22.70	22.91

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

[3] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number EPFL-CONF-192376, 2011.

[4] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[5] Kisuk Lee, Jonathan Zung, Peter Li, Viren Jain, and H Sebastian Seung. Superhuman accuracy on the snemi3d connectomics challenge. *arXiv preprint arXiv:1706.00120*, 2017.

[6] Aleksandar Zlateski and H Sebastian Seung. Compile-time optimized and statically scheduled n-d convnet primitives for multi-core and many-core (xeon phi) cpus. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 8:1–8:10. ACM, 2017.