Optimizing CPU Performance for Convolutional Neural Networks

Firas Abuzaid
Stanford University
fabuzaid@cs.stanford.edu

Abstract

We hypothesize and study various systems optimizations to speed up the performance of convolutional neural networks on CPUs. Currently, large-scale CNN experiments require specialized hardware, such as NVidia GPUs, and specialized APIs, such as NVidia's CuDNN library, to achieve adequate training performance. This provides a significant barrier to research, as the availability of high-performance GPUs is rather scarce. To address this problem, we examine the possibility of optimizing CNN performance for CPUs by borrowing techniques that have previously been used for GPUs and studying the various performance trade-offs therein. With these improvements, we are able to train on CPUs up to 4X faster than state-of-the-art systems, such as Caffe, while still maintaining statistical correctness.

1. Introduction

Deep convolutional neural networks (CNNs) have emerged as one of the most promising techniques to tackle large-scale learning problems in computer vision. Within these networks, a series of convolutional layers are typically used to extract translation invariant features from the image, a very useful property. A limiting factor, however, for using convolutional nets on large data sets was, until recently, their computational expense. This changed in 2012, when Krizhevsky et al. demonstrated that training large CNNs with millions of weights and massive data sets was computationally tractable on GPUs [8], which subsequently sparked significant interest in various other CNN frameworks and libraries that incorporated GPU support, including Torch [4], Theano [1], cuda-convnet (Krizhevsky (2014)) and Caffe [6]. Many of these frameworks are based around the CUDA library, a set of APIs that are compatible with NVidia GPUs [5].

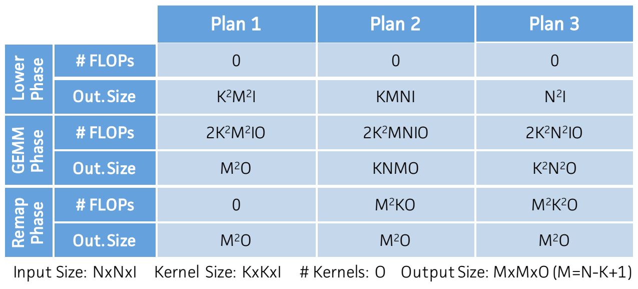
Thus far, optimizing the convolution operation for CPU performance has proven to be elusive, and the efficiency of the convolutional layer is critical for training these networks, because the convolutional layer is often the bottleneck in these computations. This is despite the fact that, in a typical CNN, the convolutional layers may only have a small fraction (i.e. less than 5%) of the parameters. However, at runtime, the convolution operations are computationally expensive and take up about 67% of the time; other estimates put this figure around 95% [7]. This makes typical CNNs about 3X slower than their fully connected equivalents (assuming equal size) [2].

Many of the optimizations employed in CuDNN [3] can also be applied to enhance the performance of CNNs on CPUs, specifically for the convolutional layers. With these improvements for convolution operations, we can train 4X faster than state-of-the-art systems, such as Caffe. Currently Caffe's CPU code does not batch the convolutional computation for multiple images; rather it computes the convolution for a single image via a single matrix multiplication. However, as is demonstrated in Chetlur et al. (2014) and Garland et al. (2008) [3], [5], the convolution can be calculated for each image separately by "lowering" the $N \times C \times H \times W$ 4D input data tensor and the $K \times C \times R \times S$ filter weight tensor into respective 2-dimensional representations, and then using a matrix multiplication operation with a corresponding kernel matrix to compute the correct result.

To demonstrate this lowering strategy, we created a new framework called Caffe ConTroll (CcT), a controller or optimizer for Caffe. In CcT, we demonstrate that this lowering strategy is also effective for CPU performance; furthermore, we examine the ability to lower the 4D data and filter tensors in various other ways, rather than the straightforward lower technique used by Chetlur et al. (2014). These different lowering approaches actually a series of tradeoffs between memory bandwitdh and computational cost, which vary across various dimensions of $N, C, H, W, K, R,$ and S. By employing these techniques and analyzing these tradeoffs intelligently, we are able to train large-scale CNNs on CPUs much more efficiently.

2. Lowerings

We introduce three different types of lowerings as shown in Figure 1. Plan 1 refers to the classical plan used by



		Plan 1	Plan 2	Plan 3
Lower Phase	# FLOPs	0	0	0
	Out. Size	K^2M^2I	KMNI	N^2I
GEMM Phase	# FLOPs	$2K^2M^2IO$	2K²MNIO	$2K^2N^2IO$
	Out. Size	M^2O	KNMO	K^2N^2O
Remap Phase	# FLOPs	0	M^2KO	M^2K^2O
	Out. Size	M^2O	M^2O	M^2O

Input Size: NxNxI Kernel Size: KxKxI # Kernels: O Output Size: MxMxO (M=N-K+1)

Figure 1. The cost model for each of the three lowering techniques. The lowerings break down into three different phases: the lowering phase, the GEMM phase, and the remap phase.

Chetlur et al. in the CuDNN library. Plan 2 uses a different approach – we do no convolutional "windowing" prior to executing the matrix multiplication. In contrast, Plan 1 applies the 2D convolution windowing algorithm to the input data tensor. In Plan 2, the convolutional window is applied after the GEMM output.

Plan 3 represents a compromise of sorts between Plan 1 and Plan 2. In Plan 3, we apply a 1D convolution window across the width dimension W to the input data tensor before the matrix multiplication step. After the GEMM output, we apply another 1D convolution along the height dimension H. The cost models for each of these lowering plans are summarized in Figure 1.

3. Experiments

In order to confirm our hypothesis – that our hardware efficiency is superior to Caffes while still maintaining the same statistical efficiency – we conducted two sets of experiments. These two experiments are both necessary in order to confirm that we have the same statistical efficiency as Caffe independently from any experiment that demonstrates better hardware efficiency.

To confirm statistical efficiency, we examined the number of iterations vs. the measured loss per iteration on a variety of networks, such as LeNet on MNIST data set and AlexNet on the ImageNet data set. Our experiment protocol was as follows: we ran each of these networks both on Caffe and on CcT and then measured the loss after each iteration. We expected to see that, within a reasonable amount of error, the measured loss for each system converge at roughly the same rate. Then, to examine hardware efficiency, we examined the throughput, both on a per-layer basis and on a per-iteration basis. Finally, we conducted these experiments across multiple lowering strategies to show the tradeoffs between these different approaches.

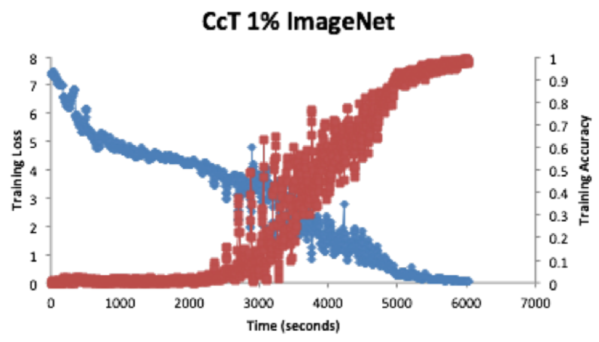
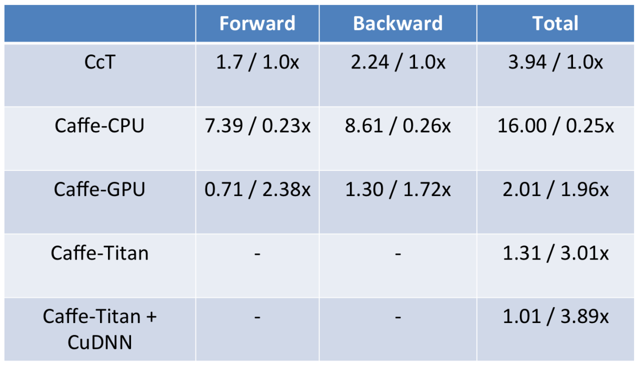


Figure 2. The training loss and training accuracy of CcT on AlexNet. This was run on 1% of the ImageNet dataset. As is shown, the loss clearly converges after several thousand iterations, and the training increases as a result.



	Forward	Backward	Total
CcT	1.7 / 1.0x	2.24 / 1.0x	3.94 / 1.0x
Caffe-CPU	7.39 / 0.23x	8.61 / 0.26x	16.00 / 0.25x
Caffe-GPU	0.71 / 2.38x	1.30 / 1.72x	2.01 / 1.96x
Caffe-Titan	-	-	1.31 / 3.01x
Caffe-Titan + CuDNN	-	-	1.01 / 3.89x

Figure 4. An overall comparison of the execution for all convolutional layers for a single iteration of AlexNet between CcT, Caffe-CPU, and Caffe-GPU. CCT and Caffe-CPU ran on a single 8-core Haswell @ 2.9GHz. Caffe-GPU ran on a single NVidia K520 without CuDNN API support. Caffe-Titan estimated from Caffe's public benchmark. Caffe-Titan + CuDNN estimated from Caffe's public benchmark.

4. Results

Our results show that, with these lowering techniques we are able to comfortably beat Caffe's CPU performance by a factor of 4. A summary of these results can be seen in 4. More importantly, our throughput for the convolutional layers is within a factor of 4 of Caffe's ideal GPU performance on the NVidia GeForce Titan with CuDNN support. This indicates that, in terms of monetary cost, we should potentially reconsider how much we rely on GPUs for large-scale CNN training.

We also were able to demonstrate that our system also achieves the same statistical correctness as Caffe; using CcT instead of Caffe will provide the same model weights after any number of training iterations (within a reasonably tolerable amount of noise). Figure 2 demonstrates that CcT

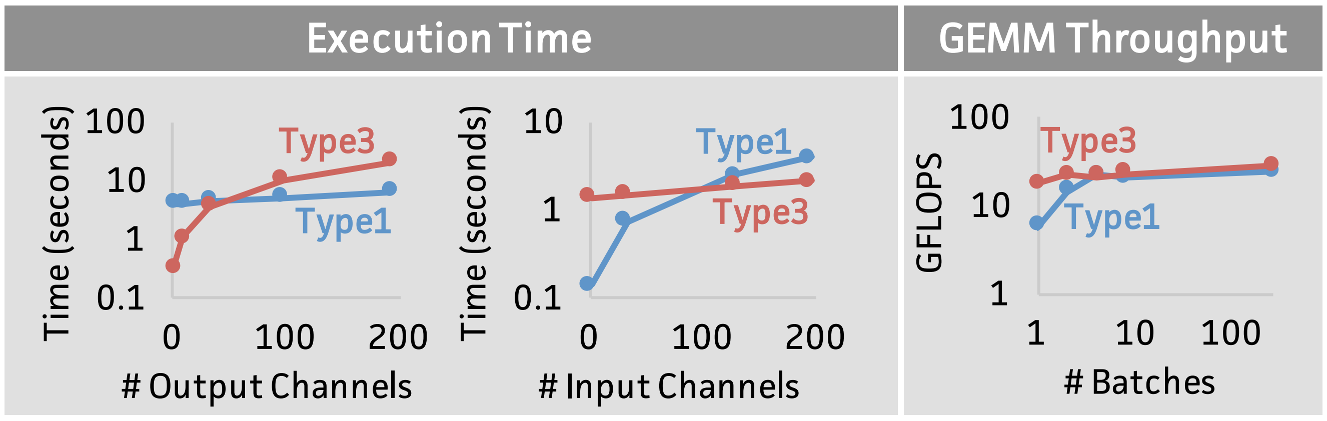


Figure 3. This illustrates the tradeoff curve between Plan 1 and Plan 3 of the different lowering strategies. Plan 2's implementation was significantly slower, so we did not include it in this graph. One area of future work that we'd like to focus on is optimizing Plan 2 in order to examine its potential benefits within this tradeoff space.

does in fact converge on the 1% ImageNet dataset at the same rate as Caffe.

In terms of the trade-offs between the different lowering strategies, our results show that when the number of output channels is smaller than the input channels, Plan 3 outperform Plan 1. However, when the number of output channels is larger than the input channels, Plan 3 pays a large cost in generating a much larger result by the matrix multiplication kernel. These trade-offs can be seen in Figure 3.

Lastly, we examined the degree of parallelism of our system by focusing on the potential speed-up when increasing the number of threads and the batch size during training. For the batch size, there is a significant trade-off here: memory consumption increases linearly with batch size. As shown in Figure 6, we see diminishing returns around 8 threads and a batch size of 100. This is somewhat surprising, as the RAM consumption for a batch size of 100 is still less than 2 GB. This is something that we'd like to further examine in the future; the most likely culprit of this result is an unintillegent strategy in parallelizing between batches of various sizes. Currently, we only adjust the number of threads at compile-time, rather than at run-time, which means we may not be partitioning the batch size effectively.

5. Future Work

In addition to the problems we previously identified, we'd like to explore several other optimizations that we believe can further improve our system. Currently, CcT only supports a particular lowering plan at compile-time; we'd like to improve this by designing a more intelligent system that *chooses* the appropriate lowering plan at run-time instead. We currently also have an un-optimized version of the Plan 2 lowering that yields very poor throughput; this needs to be optimized so that we can examine its trade-offs against Plans 1 and 3.

In terms of broader improvements, we'd like to explore other strategies for improving convolutional performance for both CPUs and for GPUs. For GPUs, recent work has shown that the Fast Fourier Transform may yield significant improvements compared to current frameworks [9]. We can also improve our approach to parallelism; currently, we only use data parallelism to improve our performance, but recent work in the literature suggests that that may not necessarily be the best approach. For example, while data parallelism may work well for the convolutional layer (due to its limited set of free parameters), this may not work well for fully-connected layers. For the latter case, model parallelism may be a better approach [7]. We wish to further explore this and integrate it as part of CcT.

References

[1] 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 compiler in python.

[2] K. Chellapilla, S. Puri, and P. Simard. High Performance Convolutional Neural Networks for Document Processing. In G. Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), Oct. 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[3] 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.

[4] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[5] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with cuda. *IEEE micro*, (4):13–27, 2008.

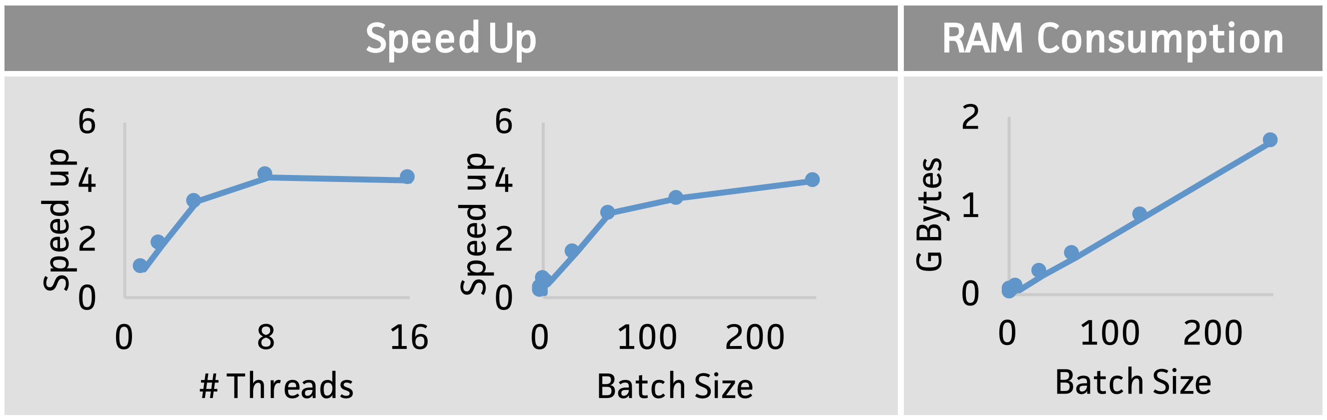


Figure 6. The effects of batch size and number of threads on the throughput of the GEMM kernel. This was run on 8 physical cores, which could potentially explain the drop-off in throughput after 8 threads.

[6] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[7] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

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

[9] 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.

Layer	CcT	Caffe-CPU	Caffe-GPU
Conv2 forward	481	1109	145
Conv1 forward	260	964	157
Conv4 forward	197	781	134
Conv5 forward	162	606	101
Pool1 forward	144	583	11
Conv3 forward	140	517	94
Pool2 forward	128	344	8
Pool5 forward	50	78	2
Fc6 forward	43	43	19
Norm1 forward	35	1277	6
Norm2 forward	28	831	4
Relu1 forward	28	73	6
Relu2 forward	24	47	4
Fc7 forward	20	20	9
Relu3 forward	13	16	1
Relu4 forward	13	17	1
Relu5 forward	10	11	1
Loss forward	8	10	8
Fc8 forward	5	5	2
Relu6 forward	2	1	0
Drop7 forward	2	8	0
Drop6 forward	2	8	0
Relu7 forward	1	1	0

Figure 5. The per-layer forward pass execution time (ms) for a single iteration of AlexNet. Plan 1 was used as the lowering strategy. Layers ordered by longest runtime in CcT.