Understanding the Performance of Small Convolution Operations for CNN on Intel Architecture

Alexander Heinecke, Evangelos Georganas, Kunal Banerjee, Dhiraj Kalamkar, Narayanan Sundaram, Anand Venkat, Greg Henry, Hans Pabst
Intel Corporation

ABSTRACT

Convolution layers are prevalent in many classes of deep neural networks, including Convolutional Neural Networks (CNNs) which provide state-of-the-art results for tasks like image recognition, natural language processing, and speech recognition. The computationally expensive nature of a convolution operation has led to the proliferation of implementations including matrix-matrix multiplication formulation, FFT-formulation, Winograd transformation, and direct convolution primarily targeting GPUs. In this paper, we optimize a direct convolution and Winograd implementation for x86 architectures, in particular for Xeon Phi systems, via a dynamic compilation approach. We then show how these JIT optimizations can be integrated in a high-level domain-specific language setting. We shed light on what is possible and what is not possible based on different data-formats and blocking techniques. Our JIT-based Ninja implementation shows close to theoretical peak results on modern x86 architectures, depending on setting and the CPU architecture at hand.

KEYWORDS

deep learning, Intel Architecture, convolution, vectorization

ACM Reference format:
Alexander Heinecke, Evangelos Georganas, Kunal Banerjee, Dhiraj Kalamkar, Narayanan Sundaram, Anand Venkat, Greg Henry, Hans Pabst. 2017. Understanding the Performance of Small Convolution Operations for CNN on Intel Architecture. In *Proceedings of ACM SC17 Conference, Denver, Colorado, USA, November 2017 (SC'17),* 2 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND OVERVIEW

In the last two years, deep learning has developed into one of the most important computational concepts. Several academic groups, and companies, have released open source frameworks which abstract many implementation details from the data scientist: TensorFlow [1], Caffe [3], to mention the most popular ones according to GitHub stars. Although these different frameworks may emphasize different workloads, one of the most important application scenario of neural networks is image recognition, [4]. This is implemented via so-called convolutional neural nets (CNN). Layers of widely-used network topologies are based on small convolutions which can be easily mapped onto CPUs and GPUs via library functions.

For direct convolutions, meta-programming via templates or static compilation (e.g. [6]) are often employed to achieve close to peak performance on a given architecture. This approach not only imposes a static compilation step, but also have to be tuned for each topology separately. Often kernel libraries fall back to hand-optimized assembly code to squeeze the last bit of performance for standard benchmark topologies, e.g. Alexnet [4].

2 IMPLEMENTATION

Prior work by [2] has shown that statically-tuned BLAS-calls incur overheads for small GEMMS and therefore do not achieve the highest performance on x86. They propose to use runtime code specialization via a JIT for small GEMMs and achieve peak performance. We employ a similar JIT strategy to implement fast direct convolutions on CPUs in this paper. We lay out the convolution data for input, output, and filter in a vectorization and cache friendly manner, and apply standard compiler optimizations such as register and cache blocking. Some of the key optimizations we apply include the software prefetching for Xeon Phi systems and the code size reduction techniques to fit all specialized versions in L1 instruction cache. We demonstrate that peak performance can be achieved on x86 systems using this approach. One thing to keep in mind is that our JIT does not incur the overheads of recompilation and tuning. Our code is available open source under https://github.com/hfp/libxsmm/.

2.1 Kernel Generation

The input parameters to convolution including N, C, K, H, W, R, S, u, and v vary significantly across benchmarks, making it almost impossible to achieve peak performance via static compilation. For example, the loops to tile and their tiling factors can not be determined statically without knowing the actual values of the parameters. Since these values are only known during the execution of a neural network, we propose a runtime code specialization approach. Beside of an actual Just-In-Time approach (lazy), an ahead-of-time compilation (at runtime) is sufficient for all of the popular machine learning frameworks, which defeats any latency issue during the execution of the network. At runtime, we apply special data formats for input/output/weight data, compiler optimizations such as tiling and register blocking to optimize the seven loop nests, runtime code specialization, and software prefetching. This makes the Ninja code for direct convolution in x86.

2.2 Kernel Streams

In the innermost loops (W, R, S) we call the proper high-performance JIT-ed kernel that takes six arguments: the addresses for the input, weight and output blocks to be convoluted in the

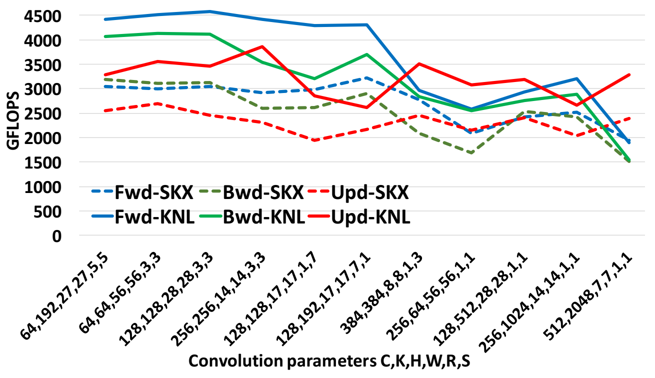

Figure 1: Direct convolution performance

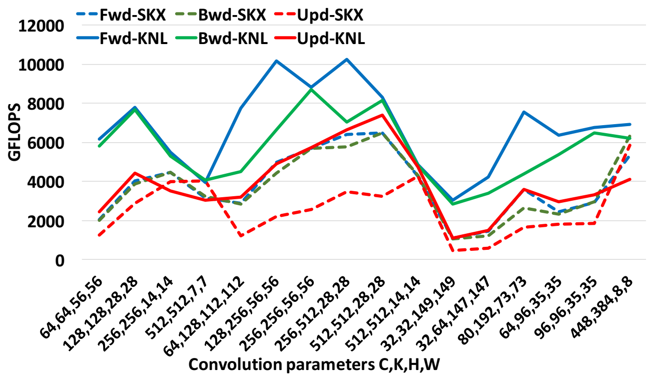

Figure 2: Winograd convolution performance

current iteration and the addresses for the input, weight and output blocks to be prefetched for the following iteration. This approach exhibits two performance impediments. First, the address calculations of the corresponding tensor blocks involve integer multiplications and additions. Second, calculating the addresses of the tensor blocks to be prefetched entails complicated conditional statements. We alleviate these two issues by developing a technique we call *kernel streams*. During the JIT-ing phase we perform a dry run of the convolution loops and we compute streams of address offsets for the kernel call arguments. These streams are computed on a per-thread basis. Subsequently, the actual convolution run is just a replay of offsets additions to base addresses and kernel calls in a simple loop.

2.3 Winograd

For small 3x3 filters, Winograd based convolution is found to be superior to direct convolutions [5]. Note that Winograd can be thought of as a special case of FFT involving similar transformations between time domain and frequency domain. The arithmetic complexity of the multiplication is: $N \cdot (H/m) \cdot (W/n) \cdot C \cdot K \cdot (m + R - 1) \cdot (n + S - 1)$, where m and n are the height and width of a tile of a transformation block, $m = n = 1$ results in the previously discussed direct variant. For a tile size of 6x6 the arithmetic complexity can be therefore significantly reduced at the price of adding bandwidth bound transformations. By transforming into frequency space, Winograd allows to replace the convolution operation by a point-wise multiplication which can be formulated as a batched GEMM when being blocked over channels and/or images of the minibatch. For best performance this operation is accelerated by a JIT approach as well.

3 PERFORMANCE SUMMARY

Figures 1 and 2 depict the performance of LIBXSMM on a single socket of Intel Xeon Phi 7250 (KNL) with 68 cores and a single socket of Intel Xeon Platinum 8180 (SKX) with 28 cores. KNL offers a SGEMM peak performance of 4.6 TFLOPS, whereas SKX delivers roughly 3.2 TFLOPS for the same benchmark.

In case of direct convolutions, 5x5 and 3x3 are able to achieve close to SGEMM peak on both platforms, whereas 1x1 convolutions are a bit slower. In case of SKX, the large data caches help to achieve high performance although no high bandwidth memory is present. Winograd is able to deliver an up to 2.2× speed-up over direct 3x3 convolutions which is expected, based on saved operations but additionally needed transformations.

In addition to pure kernel optimizations, our work consists of contributions, which has been subsequently released since TensorFlow v1.1, and changes supporting this publication (but meant to become part of TensorFlow). For an end-to-end run of Tensorflow with LIBXSMM employing the Inception v3 model we measured a speed-up of 1.5× for inference and 1.1× for training over vanilla Tensorflow.

4 CONCLUSION AND FUTURE WORK

The current status of the work demonstrates that it is possible to implement a highly-efficient small convolution operations for Intel processors. This is true for both scenarios: memory bandwidth bound and compute bound usage. However, there are still design parameters which can be even further optimized. Along these lines auto-tuning of e.g. the composition of different micro-kernels (e.g. $M = 24$ can be built by 16+8 or 12+12) is a future research direction. Another direction is to take kernel streams and apply this technique to other domains where intercepting standard interfaces is of great value, and helps to exploit hardware capabilities by the means of processing batches of similar calls.

REFERENCES

[1] Martín Abadi and others. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/
[2] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation *(SC '16)*. Article 84, 11 pages.
[3] Yangqing Jia and others. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).
[4] Alex Krizhevsky, I. Sutskever, and G.E. Hinton. 2012. Image Classification with Deep convolutional neural networks. *Advances in neural information processing systems* (2012), 1097–1105.
[5] Andrew Lavin and Scott Gray. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015). http://arxiv.org/abs/1509.09308
[6] Nervana Systems. 2016. NEON. https://github.com/NervanaSystems/neon. (2016).