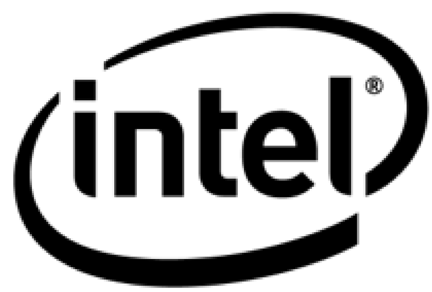


Understanding the Performance of Small Direct Convolution Operations for CNN on Intel Architecture

Alexander Heinecke[1], Evangelos Georganas[1], Kunal Banerjee[2], Dhiraj Kalamkar[2], Narayanan Sundaram[1], Anand Venkat[1], Greg Henry[3], Hans Pabst[4]

[1]Intel Corporation, Intel Labs, Mission College Boulevard 2200, Santa Clara 95054, CA, USA
[2]Intel Technology India Private Limited, Intel Labs, #23-56P, Devarabeesana Halli, Outer Ring Road, Bangalore-560103, India
[3]Intel Corporation, Software and Services Group, 2111 NE 25th Avenue, Hillsboro 97124, OR, USA
[4]Intel Semiconductor AG, Software and Services Group, Badenerstrasse 549, 8048 Zurich, Switzerland

Introduction

LIBXSMM is a library targeting Intel Architecture for small, dense or sparse matrix multiplications, and small convolutions. LIBXSMM is available as free software at https://github.com/hfp/libxsmm.

- Work targets Convolutional Neural Networks (CNNs), and other deep neural networks.
- Convolution operators are thread-library agnostic (pthreads, OpenMP, C++ threads, Cilk, etc.).
- Direct convolution and Winograd transformation implemented.
- Convolution layer is integrated into TensorFlow.
- Optimized for x86 (Xeon Phi and Xeon server).

This work leverages LIBXSMM's infrastructure to generate executable code Just-In-Time (JIT) by assembling the instructions in-memory.
For reproducible results and for general use, we show how the JIT optimizations integrate with a high-level domain-specific language such as TensorFlow.

General Design and Interface

Interface

- LIBXSMM provides `libxsmm_dnn_layer` as an opaque container for several operations such as convolution, pooling, LSTM cells, etc.
- Activations, filters, bias, etc. are represented through the opaque datatype `libxsmm_dnn_buffer` which is a wrapper for user provided pointers in a data format support by LIBXSMM.
- Without copy-in, LIBXSMM supports $NHWC/RSCK$ (Tensorflow's native layout) and its custom formats (see below). $NCHW/KCRS$ (preferred cuDNN layout) are supported per copy-in.
- The `libxsmm_dnn_layer` and `libxsmm_dnn_buffer` are then linked together via API calls to create and executable layer operation.
- To build topologies, one can create either buffers on each layer or use LIBXSMM's buffers to define layer connections. As buffers are just pointer wrappers of user data there is no performance difference between both approaches. This allows an easy and local integration of LIBXSMM into frameworks.

Data Layouts

- CPUs favor vectorizations without a need for gather/scatter, for convolutions operations that means only 3 out of 7 dimension are suited: N, C, K.
- LIBXSMM supports therefore four data layouts, where generally $c = k = n$ and they match the vector-length of the underlying CPU architecture (here 16 for AVX-512 and FP32):
  - $NBHWk/BDRSck$ with $K = Bk$ and $C = Dc$
  - $NHWBk/BDRSck$ with $K = Bk$ and $C = Dc$
  - $NHWBk/RSDcBk$ with $K = Bk$ and $C = Dc$
  - $BYHWkn/BDRSck$ with additinonaly $N = Yn$
- Winograd internally uses a different layout due to its tiling, but the frontend is available for $NBHWk/BDRSck$ and $NHWBk/BDRSck$.
- Our Tensorflow integration uses $NHWBk/BDRSck$ and we work on modified filters to avoid conflict misses.
- For this poster, we use $NBHWk/BDRSck$ as it exhibits best spatial and temporal locality.

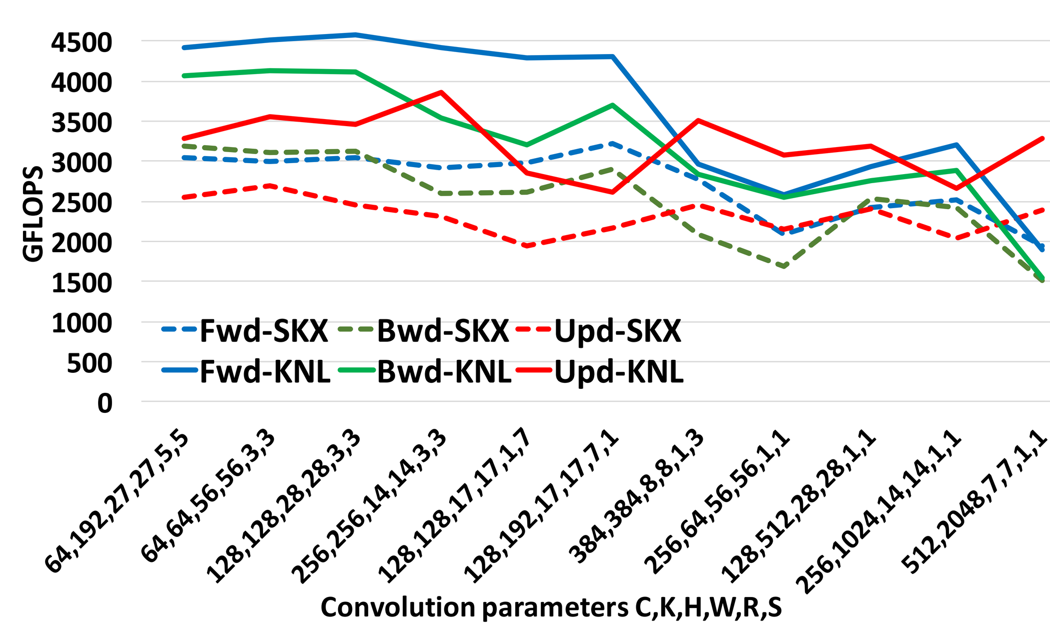
Performance

We have evaluated the performance in the context of standalone convolution operators using direct convolutions and Winograd for 3x3 convolutions. We selected the most important layer operations from Alexnet, Resnet-50 and Google's Inception v3 topologies. The later two are the most modern topologies in production.

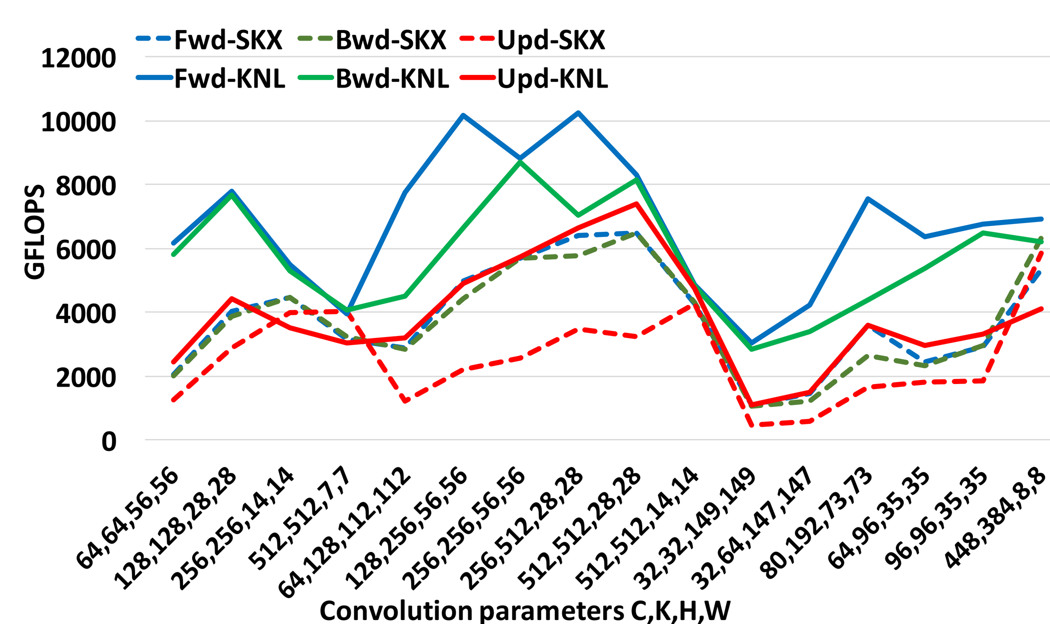
We cover both inference and training operations: Fwd: forward convolutions, Bwd: backward activation convolution, Upd: backward filter convolution. Fwd is needed for inference and training, whereas Bwd and Upd are only required for training.

Our test platform is a single-socket Intel Xeon Platinum 8180 (SKX) with 28 cores and a Intel Xeon Phi 7250 (KNL) with 68 cores. On both platforms Intel Turbo Technology was enabled. On the Xeon Platinum SGEMM runs at 3.2 TFLOPS where as the Xeon Phi 7250 delivers 4.6 TFLOPS for SGEMM. We also present measurements on a Knights Mill processor (skipped for submission). All measurements are based on Version 1.9 of LIBXSMM, https://github.com/hfp/libxsmm/releases.

Direct Convolutions



Winograd



Discussion

- For 3x3 and 5x5 convolutions, all platforms deliver their SGEMM performance for forward convolutions.
- Bwd and Upd have substantially short accumulation chains (pure FMA instruction code streams, SGEMM-like execution) which leads to a drop in performance on KNL.
- SKX can maintain this high performance for all sizes and convolution directions.
- Minimal decrease in performance on SKX for Bwd and Upd is caused by data transformations needed and which are incl. in the timings.
- Winograd can deliver an up to 2.2× boost (KNL).
- Arithmetic operation savings of Winograd are as high as 4×, however they have to be paid for by bandwidth-bound transformations.
- Therefore boost is higher on KNL than on SKX due to availability of high bandwidth MCDRAM.

Implementation Details

Kernel Streams

- Convolutions consist of seven nested loops, each one corresponding to the parameters N, C, K, H, W, R, S. In the innermost loop we call the JIT-ed kernel.
- The kernel takes six arguments: 3 addresses for the input, weight and output blocks to be convoluted in the current iteration and 3 addresses for the input, weight and output blocks to be prefetched for following iteration.

This approach exhibits two overheads during execution:

- The address calculations of the tensor blocks involve integer multiplications and additions.
- Calculating the addresses of the tensor blocks to be prefetched entails complicated conditional statements.

We develop a technique we call *kernel streams* to alleviate these overheads:

1. 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.
2. The actual convolution run is just a replay of offsets additions to base addresses and kernel calls in a simple loop.

Winograd

- The implementation is split into 4 kernels: input transform, weight transform, batched GEMM, output transform.
- For best performance we implemented a JIT-based batched GEMM.

Integration into Tensorflow

- LIBXSMM leverages Tensorflow's thread pool and offers native support for TF's data format (which eliminates costly data transformations).
- Current integration speeds up Tensorflow by 1.5× running Inception v3 model on an Intel Xeon Platinum CPU.

Summary/Outlook

LIBXSMM ...

- ... achieves optimal performance for wide range of convolution operators on latest Intel Xeon Platinum (code-named Skylake) and Intel Xeon Phi (code-named Knight Landing) processors.
- ... integrated into Tensorflow and leverages Tensorflow's execution model.
- ... already offers support for Intel's Machine Learning oriented chip code-named Knights Mill.
- ... is freely available and includes examples for usage.

Current Research:

- Adding a runtime auto-tuning component for both dispatching and micro-kernel composition.
- Providing LSTM/RNN modules based on small blocked GEMMs.

References

[1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[2] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. Libxsmm: Accelerating small matrix multiplications by runtime code generation. SC '16, pages 84:1–84:11, 2016.

[3] Yangqing Jia et al. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[5] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. *CoRR*, abs/1509.09308, 2015.

[6] Nervana Systems. NEON. https://github.com/NervanaSystems/neon, 2016.