High-Performance Deep Learning via a Single Building Block

Evangelos Georganas, Kunal Banerjee, Dhiraj Kalamkar, Sasikanth Avancha, Anand Venkat, Michael Anderson, Greg Henry, Hans Pabst, Alexander Heinecke
Intel Corporation

ABSTRACT

Deep learning (DL) is one of the most prominent branches of machine learning. Due to the immense computational cost of DL workloads, industry and academia have developed DL libraries with highly-specialized kernels for each workload/architecture, leading to numerous, complex code-bases that strive for performance, yet they are hard to maintain and do not generalize. In this work, we introduce the *batch-reduce GEMM kernel* and show how the most popular DL algorithms can be formulated with this kernel as the basic building-block. Consequently, the DL library-development degenerates to mere (potentially automatic) tuning of loops around this sole optimized kernel. By exploiting our new kernel we implement Recurrent Neural Networks, Convolution Neural Networks and Multilayer Perceptron training and inference primitives in just 3K lines of high-level code. Our primitives outperform vendor-optimized libraries on multi-node CPU clusters, and we also provide proof-of-concept CNN kernels targeting GPUs. Finally, we demonstrate that the batch-reduce GEMM kernel within a tensor compiler yields high-performance CNN primitives, further amplifying the viability of our approach.

1 INTRODUCTION AND RELATED WORK

In the past decade, machine learning has experienced an academic and industrial renaissance where deep learning (DL) has been the main driving force. More specifically, deep neural networks have advanced the fields of computer vision, speech recognition, machine translation and search ranking, and naturally emerge in numerous applications and scientific domains [1–6].

Three types of neural networks (NN) comprise the most prominent DL workloads by representing 95% of the data-centers's demands [7]: i) Recurrent Neural Networks (RNN) [8] with the so-called Long Short-Term Memory (LSTM) [9] networks being the most popular variation, ii) Convolution Neural Networks (CNN) [1], and iii) Multi-Layer Perceptrons (MLP) [10, 11]. Additionally, the contemporary Transformer [12] and BERT [13] workloads computationally involve fully-connected layers which also lie in the heart of MLP. Due to the increase of the involved datasets' size and complexity in deep neural networks (DNN), they require vast amount of computation. Therefore, academia and industry have invested into the development of DL libraries targeting all the aforementioned workloads on various architectures.

The development of such DL libraries typically embraces one of the following strategies: (i) the specific workload kernel leverages coarse-grained, linear algebra library calls, e.g. LSTM cell via large GEneral Matrix Multiply (GEMM) calls in mkl-dnn [14], convolutions via image-to-column tensor transformations and subsequent large GEMM calls [15, 16], or (ii) for each workload and use-case (training/inference) the kernel employs a specialized implementation that targets the specific algorithm/workload and architecture

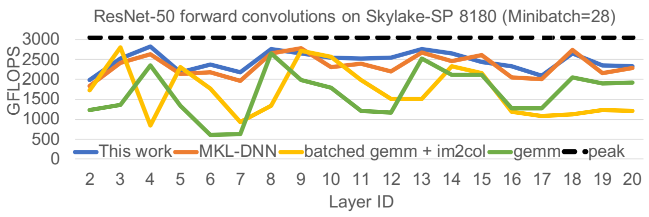

Figure 1: Performance of ResNet-50 forward convolutions

at hand, e.g. convolution kernels in mkl-dnn and cuDNN [17]. The former approach of deploying coarse-grained, linear algebra library calls provides ease in the DL library development process since no special kernel development is involved. However it may result in suboptimal data reuse (e.g. redundant data movements to format underlying tensor/matrices in the required layout that enables GEMM calls), and also it is not flexible enough to allow efficient, fine-grained fusion of other operators. The latter approach of implementing specialized kernels for each DL workload/use-case and platform/architecture strives for performance but naturally results in numerous, complex code-bases that are hard to maintain and do not generalize. For example, the code-base *only for convolutions on CPUs* within mkl-dnn consists of ~36,000 lines of code. Figure 1 shows the performance of various convolution kernel implementations on a Xeon Skylake-SP 8180 processor. The yellow and green lines represent implementations adopting strategy (i). More specifically, the green line shows the performance of convolutions that leverage small GEMM library calls, whereas the yellow line illustrates the performance of an implementation which uses image to column transformations and *batched GEMM* [18] library calls. Both approaches perform far from the machine's peak with average efficiencies of 61% and 49% respectively. On the other hand, the orange line exhibits the performance of the vendor-optimized mkl-dnn library that follows strategy (ii) with ad hoc, specialized direct convolution kernels and achieves average efficiency of 81%, being 1.33× and 1.64× faster than the aforementioned generic implementations. However, this performance comes at the cost of complex, specialized kernels that do not generalize to different workloads (e.g. RNN/LSTM/MLP) or different architectures (e.g. GPUs).

In this work, we introduce a new kernel called *batch-reduce GEMM* and show how the most popular DL workloads and algorithms (RNN/LSTM, CNN and MLP) can be formulated with this new kernel as basic building block. The batch-reduce GEMM kernel essentially multiplies a sequence of input sub-tensor blocks (which form a *batch*) and the partial multiplication results are *reduced* into a single accumulator/output sub-tensor block. Our new kernel is flexible enough to accommodate coarse-grained and fine-grained operations that arise in DL workloads, whereas its semantics lend themselves to various optimizations (e.g. load/store optimizations of the result sub-tensor, prefetching of the sub-tensors to be multiplied). Also, since the kernel supports operations at fine granularity, fusion of subsequent operators on the output sub-blocks is inherently efficient. The blue line in Figure 1 shows the performance of the convolution primitive that leverages our new *batch-reduce GEMM* kernel achieving average efficiency of 83%, and outperforms even the ad hoc, vendor-optimized kernel.

Having a single kernel as basic building-block is transformative: by implementing and optimizing this single kernel for a given architecture, the development of DL primitives degenerates to mere loop tuning around this kernel. Essentially our approach with a *single* kernel addresses the issue of combinatorial explosion of low-level optimization work that is required for each pair <architecture, DL primitive>. Instead, for each architecture we need to optimize at low-level *only one kernel for all DL primitives.*

Furthermore, having a single, highly efficient building-block enables efficient usage of tensor compiler frameworks. Such frameworks embrace tensors as first class citizens, and provide specific optimization techniques targeting tensor algebra programs. Since DL primitives are inherently tensor algebra programs, there is a large amount of ongoing research that leverages specialized tensor compilers for DL workload development (e.g. TVM [19], GLOW [20], PlaidML [21], MLIR [22]). However, compilers struggle to optimize small GEMM-flavored loop nests that arise in tensor programs [23]. Our kernel is optimized for the nuances of the architecture at hand, and serves tensor compilers a robust building block that can be used during the polyhedral optimization phase of general loop nests [21, 24].

To illustrate the viability and generality of our methodology with a single kernel, we develop DL primitives which target training and inference of RNN/LSTM, CNN and MLP workloads in ~3,000 lines of high-level C code. Our primitives outperform vendor-optimized libraries on CPUs. We also provide proof-of-concept design with a tensor compiler framework by showcasing efficient CNN implementation in TVM that leverages our batch-reduce GEMM kernel. Additionally, our methodology provides a pathway for performance portability; we present exemplary, high-performance CNN kernels on integrated GPUs. The main contributions of this poster are:

- The introduction of the batch-reduce GEMM kernel along with its efficient implementation.
- The design and implementation of multi-threaded, high performance DL primitives covering RNN/LSTM, CNN and MLP inference and training algorithms with batch-reduce GEMM kernel being the basic building block. We need to optimize at low-level *only this kernel for all DL primitives.*
- A detailed performance comparison of our DL primitives with state-of-the-art vendor-optimized libraries.
- CNN proof-of-concept results on integrated GPUs and CNN kernels within TVM that leverage the batch-reduce GEMM kernel.

2 THE BATCH-REDUCE GEMM KERNEL

In this section, we describe the design and implementation of the new batch-reduce GEMM kernel which comprises the cornerstone of our deep learning primitives. The functionality of the new kernel materializes the operation:

$$C_j = \beta \cdot C_j + \alpha \sum_{i=0}^{N-1} A_i \cdot B_i$$

Algorithm 1 The batch-reduce GEMM kernel

```
Inputs: $A_i \in \mathbb{R}^{m\times k}, B_i \in \mathbb{R}^{k\times n} i = 0, ..., N\text{-}1, C_j \in \mathbb{R}^{m\times n}\ \alpha, \beta \in \mathbb{R}$
Output: $C_j = \beta \cdot C_j + \alpha \sum_{i=0}^{N-1} A_i \cdot B_i$
1: for $i_n = 0 \dots n-1$ with step $\mathrm{n_b}$ do
2:   for $i_m = 0 \dots m-1$ with step $\mathrm{m_b}$ do
3:     acc_regs ← load $m_b \times n_b$ $C_j$ subblock$_{i_m,i_n}$
4:     for $i = 0 \dots N-1$ with step 1 do
5:       for $i_k = 0 \dots k-1$ with step 1 do
6:         ▷ Outer product GEMM microkernel
7:         acc_regs += $A_i$ subcolumn$_{i_m,i_k}$ × $B_i$ subrow$_{i_k,i_n}$
8:     $C_j$ subblock$_{i_m,i_n}$ ← acc_regs
```

In essence, this kernel multiplies the specified blocks $A_i \in \mathbb{R}^{m\times k}$ and $B_i \in \mathbb{R}^{k\times n}$ and reduces the partial results to a block $C_j \in \mathbb{R}^{m\times n}$ of a tensor C. It is noteworthy that tensors A and B can alias and also the blocks A_i and B_i can reside in any position in the input tensors A and B. The new batch-reduce GEMM kernel takes the following arguments: (i) *two arrays of pointers* to the corresponding blocks A_i and B_i to be multiplied, (ii) a pointer to the output block C_j, (iii) the number N dictating the number of blocks to be multiplied and (iv) the scaling parameters α and β.

In order to obtain a high performance implementation of the batch-reduce GEMM kernel we build upon and extend the open source LIBXSMM [23] library which leverages JIT techniques and generates small GEMMS achieving close to peak performance. Algorithm 1 shows the pseudocode of the batch-reduce GEMM kernel. Lines 1-2 block the computation of the result C_j in $m_b \times n_b$ subblocks. Once such a subblock is loaded into the accumulation registers (line 3), we loop over all pairs A_i, B_i (line 4) and we accumulate into the loaded registers the products of the corresponding $m_b \times k$ subblocks of A_i with the relevant $k \times n_b$ subblocks of B_i (lines 5-7). In order to calculate a partial product of an $m_b \times k$ subblock of A_i with a $k \times n_b$ subblock of B_i, we follow an outer product formulation. In particular, we multiply an $m_b \times 1$ column of A_i with a $1 \times n_b$ row of B_i (line 7) and we repeat the analogous outer product computation for all k columns/rows of the A_i/B_i subblocks (line 5).

3 MAPPING BATCH-REDUCE GEMM TO DL PRIMITIVES

In this work, we show how the most popular DL algorithms (RNN/LSTM, CNN and MLP) can be formulated with batch-reduce GEMM as basic building block. We demonstrate that our methodology outperforms vendor-optimized, low-level DL primitives by factors up to 1.4×. Moreover, we integrate our DL kernels into distributed frameworks, and optimize end-to-end workflows for GNMT and ResNet-50 training. Additionally, we highlight the architectural-agnostic aspect of our methodology by matching the CNN kernel performance of a vendor-provided library on integrated GPUs. Finally, we prototype CNN kernels in a tensor compiler framework by harnessing our batch-reduce GEMM kernel, and match the performance of *auto-tuned* inference TVM primitives. As future work, we intend to experiment with Tensor compilers' automatic polyhedral optimization (e.g. [21, 24]).

REFERENCES

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

[2] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[3] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[4] Dong Yu, Michael L Seltzer, Jinyu Li, Jui-Ting Huang, and Frank Seide. Feature learning in deep neural networks-studies on speech recognition tasks. *arXiv preprint arXiv:1301.3605*, 2013.

[5] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[6] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pages 7–10. ACM, 2016.

[7] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[8] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry.* MIT press, 2017.

[11] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[14] Intel MKL-DNN. https://github.com/intel/mkl-dnn, Accessed on 4/3/2019.

[15] Aravind Vasudevan, Andrew Anderson, and David Gregg. Parallel multi channel convolution using general matrix multiplication. *arXiv preprint arXiv:1704.04428*, 2017.

[16] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. Low-memory gemm-based convolution algorithms for deep neural networks. *arXiv preprint arXiv:1709.03395*, 2017.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[18] Jack Dongarra, Sven Hammarling, Nicholas J Higham, Samuel D Relton, Pedro Valero-Lara, and Mawussi Zounon. The design and performance of batched blas on modern high-performance computing systems. *Procedia Computer Science*, 108:495–504, 2017.

[19] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[20] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018.

[21] Tim Zerrell and Jeremy Bruestle. Stripe: Tensor compilation via the nested polyhedral model. *arXiv preprint arXiv:1903.06498*, 2019.

[22] Multi-Level Intermediate Representation. https://github.com/tensorflow/mlir, Accessed on 4/10/2019.

[23] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 84:1–84:11, Piscataway, NJ, USA, 2016. IEEE Press.

[24] Roman Gareev, Tobias Grosser, and Michael Kruse. High-performance generalized tensor operations: A compiler-oriented approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):34, 2018.