High-Performance Deep Learning via a Single Building Block

Evangelos Georganas[1], Kunal Banerjee[2], Dhiraj Kalamkar[2], Sasikanth Avancha[2], Anand Venkat[1], Michael Anderson[1], Greg Henry[3], Hans Pabst[4], Alexander Heinecke[1]

[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

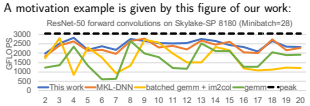
Introduction

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.

Punchline of this work is the following:

- The introduction of the JIT-compiled *batch-reduce GEMM kernel* (this work leverages LIBXSMM's (`https://github.com/hfp/libxsmm`) infrastructure) and show how the most popular DL algorithms can be formulated with this kernel as the basic building-block
- The design and implementation of multi-threaded, high performance DL primitives covering Recurrent Neural Networks (RNN/LSTM), Convolution Neural Networks (CNN) and Multilayer Perceptron (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*.
- significantly reducing the code size and code specialization and hence enhance code maintainability
- 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 demonstrating performance portability opportunities of our approach

A motivation example is given by this figure of our work:



The blue line 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. It is clearly superior over batched-gemm and classic im2col algorithms.

Properties of our proposal:

- 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
- The new kernel is flexible enough to accommodate coarse-grained and fine-grained operations that arise in DL workloads
- The new kernel supports fusion of subsequent operators on the output sub-blocks
- The new kernel is optimized for the nuances of the architecture at hand, and serves as a robust building block

These properties allow for transformative changes:

- by implementing and optimizing this single kernel for a given architecture, the development of DL primitives degenerates to mere loop tuning around this kernel
- addressing the issue of combinatorial explosion of low-level optimization work that is required for each pair <architecture, DL primitive>
- enabling efficient usage of tensor compiler frameworks such as TVM, GLOW, PlaidML, and or MLIR (DL primitives are inherently tensor algebra programs)

The Batch-Reduce GEMM Kernel

The batch-reduce GEMM kernel materializes the operation:

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** $i_n = 0 \ldots n-1$ **with step** n_b **do**
2: **for** $i_m = 0 \ldots m-1$ **with step** m_b **do**
3: acc_regs ← load $m_b \times n_b$ C_j subblock$_{i_m, i_n}$
4: **for** $i = 0 \ldots N-1$ **with step 1 do**
5: **for** $i_k = 0 \ldots k-1$ **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

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

- 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 $C_j \in \mathbb{R}^{m \times n}$
- A and B can alias and also the blocks A_i and B_i can reside at any arbitrary position

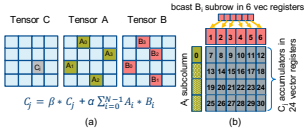
The kernel takes the following arguments:

- *two arrays of pointers* to the corresponding blocks A_i and B_i to be multiplied
- a pointer to the output block C_j
- the number N dictating the number of blocks
- the scaling parameters α and β

Algorithm 1 depicts the batch-reduce GEMM kernel:

- lines 1-2 block the computation of the result C_j in $m_b \times n_b$ subblocks
- load accumulator (line 3) and loop over all pairs A_i, B_i and subblocks (line 4-7)
- Update C_j (line 8)

The following figures highlight the micro-kernel targeting an Intel AVX-512 enabled platform (32 SIMD registers, 16 elements each):

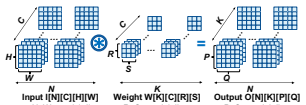


Most important properties of this kernel are:

- builds upon and extends the open source LIBXSMM library which leverages JIT techniques and generates small GEMMS achieving close to peak performance
- multiplies an $m_b \times 1$ column of A_i with a $1 \times n_b$ row of B_i (in this example $m_b = 64$, $n_b = 6$)
- accumulates into registers 7-30 that hold the partial C_j result
- broadcasts the row elements of B_i into registers 1-6
- load 16 elements of the A_i column in register 0
- performs 6 fused-multiply-add instructions (FMAs) with registers 0 and 1-6 we update the accumulators 7-12
- repeats the former for remaining 48 elements of the A_i column and we update all the accumulation registers 13-30

Note: our enhanced LIBXSMM leverages various strategies depending on the architecture at hand (i.e. vector length) and the m_b, n_b values and makes use of software prefetches for A_i and B_i elements.

Convolution Neural Networks (CNN)

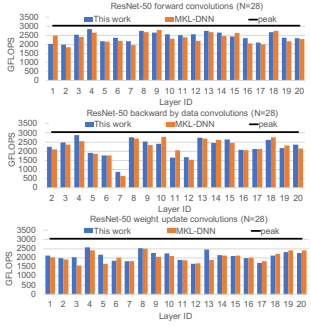


Algorithm 2 CNN forward pass via batch-reduce GEMM

1: $C_b = C/b_c$, $K_b = K/b_k$, $Q_b = Q/b_q$
2: **for** $n = 0 \ldots N-1$ **do**
3: **for** $k_b = 0 \ldots K_b - 1$ **do**
4: **for** $c_b = 0 \ldots C_b - 1$ **with step** B_c **do**
5: **for** $oj = 0 \ldots P-1$ **do**
6: **for** $oib = 0 \ldots Q_b - 1$ **do**
7: $oi = oib \cdot b_q$, $ii = str \cdot oi$, $ij = str \cdot oj$, $i = 0$
8: ▷ *Prepare batch-reduce GEMM arguments*
9: **for** $r = 0 \ldots R-1$ **do**
10: **for** $s = 0 \ldots S-1$ **do**
11: **for** $c = 0 \ldots B_c - 1$ **do**
12: $A_{ptrs}[i] = \&W[k_b][c_b + c][r][s][0][0]$
13: $B_{ptrs}[i{+}{+}] = \&I[n][c_b + c][ij + r][ii + s][0]$
14: $Out = \&O[n][k_b][oj][oi][0]$
15: **batchreduce_gemm**$(A_{ptrs}, B_{ptrs}, Out, R \cdot S \cdot B_c)$

- The input activation tensors are convoluted with the weight tensors to yield the output activation tensors.
- Batch-reduce GEMM kernel obviates the need for a specialized convolution kernel.
- The batch-reduce GEMM kernel inherently optimizes load/store of the output O in case of $R, S > 1$ and in case the input feature map loop is reordered as the innermost loop.
  - By properly selecting the sub-tensors of weights/inputs to be multiplied and reduced onto an O sub-tensor, the accumulation takes place entirely in registers
- The data reuse optimizations/parallelization tasks then simply translates to tuning the surrounding loops
- Only slight modifications to the forward kernel are required in order to implement the back-propagation kernels, as they can be mapped through linear index transformations into the forward convolution loop nest ("duality").

Performance of ResNet-50 convolutions on Xeon 8180:



- The weighted efficiency of the FWD convolutions is 83% of peak, while MKL-DNN exhibits weighted efficiency of 81% of peak (2.5% slower than our work).
- Our kernels have weighted efficiency 80% and 73.6% for the BWD pass and the UPD pass respectively, whereas MKL-DNN BWD and UPD kernels illustrate weighted efficiencies of 78.9% and 68.9% respectively, and are 1% and 7% slower than our kernels.

Long-Short Term Memory (LSTM)

Parameters: N batch-size, T sequence length, C state size and K hidden state size. The input/weight tensors are combined based on the Equations 1-6 and yield the output sequence $h \in \mathbb{R}^{T \times K \times N}$ and tensor $s \in \mathbb{R}^{T \times K \times N}$:

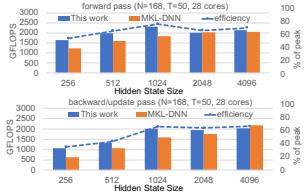
$$i_t = \sigma(W_i \cdot x_t + R_i \cdot h_{t-1} + b_i) \quad (1)$$
$$c_t = \tanh(W_c \cdot x_t + R_c \cdot h_{t-1} + b_c) \quad (2)$$
$$f_t = \sigma(W_f \cdot x_t + R_f \cdot h_{t-1} + b_f) \quad (3)$$
$$o_t = \sigma(W_o \cdot x_t + R_o \cdot h_{t-1} + b_o) \quad (4)$$
$$s_t = f_t \circ s_{t-1} + i_t \circ c_t \quad (5)$$
$$h_t = o_t \circ \tanh(s_t) \quad (6)$$

Algorithm 1 Forward propagation pass of LSTM cell

Inputs: Weight tensors $W_* [K_b][C_b][b_c][b_k]$, $R_* [K_b][K_b][b_k][b_k]$
Input sequence $x[T][N][C]$, Bias $b_*[K]$, blocking factors b_k, b_c, b_n
Outputs: Output sequence $h[T][N][K]$ and $s[T][N][K]$

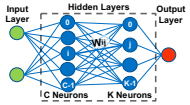
1: $N_b \leftarrow N/b_n$
2: Based on $thread\_id$ calculate K_{b}_start, K_b_end, N_{b}_start and N_{b}_end to assign output work items
3: **for** $t = 0 \ldots T-1$ **do**
4: **for** $ib_k = K_{b}\_start \ldots K_b\_end$ **do**
5: **for** $ib_n = N_{b}\_start \ldots N_b\_end$ **do**
6: ▷ *Compute a block of* $i_t = \sigma(W_i \cdot x_t + R_i \cdot h_{t-1} + b_i)$
7: $i_k \leftarrow ib_k \cdot b_k$, $i_n \leftarrow ib_n \cdot b_n$
8: $i[t][i_n][i_k] \leftarrow b_i[i_k]$
9: **for** $ib_c = 0 \ldots C_b - 1$ **do**
10: $A_{ptrs}[ib_c] = \&W_i[ib_k][ib_c][0][0]$
11: $B_{ptrs}[ib_c] = \&x[t][i_n][ib_c \cdot b_c]$
12: **batchreduce_gemm**$(A_{ptrs}, B_{ptrs}, \&i[t][i_n][i_k], C_b)$
13: **for** $ib_c = 0 \ldots K_b - 1$ **do**
14: $A_{ptrs}[ib_c] = \&R_i[ib_k][ib_c][0][0]$
15: $B_{ptrs}[ib_c] = \&h[t-1][i_n][ib_c \cdot b_k]$
16: **batchreduce_gemm**$(A_{ptrs}, B_{ptrs}, \&i[t][i_n][i_k], K_b)$
17: $i[t][i_n][i_k] \leftarrow \sigma(i[t][i_n][i_k])$
18: ▷ *Ditto for blocks of* c_t, f_t, o_t *via Equations 2-4*
19: $s[t][i_n][i_k] \leftarrow f[t][i_n][i_k] \circ s[t-1][i_n][i_k] +$
20: $i[t][i_n][i_k] \circ c[t][i_n][i_k]$
21: $h[t][i_n][i_k] \leftarrow o[t][i_n][i_k] \circ \tanh(s[t][i_n][i_k])$

- We adopt a "dataflow" based approach for optimizations
- Blocked layout to exploit locality & avoid conflict misses
- Perform computation with fused time steps
- Also, allow blocked inputs / weights to be passed directly from framework
- Use JIT batch-reduce GEMM kernels
  - Implement optimized blocked GEMM
  - Implement fused kernel for elementwise operations $(i_t, f_t, o_t, c_t, s_t, h_t)$
  - Once a block of batch-reduce GEMM is computed, apply element-wise operations on it while hot in cache
- Same optimizations applied to backward/update passes



- MKL-DNN approach: large GEMMs followed by element-wise ops – potentially poor locality for elementwise part
- For small/medium sized problems, our implementation of LSTM forward pass is up to 1.4× faster than the MKL-DNN cell, while for backward/weight update it is up to 1.3× faster on Xeon 8180.
- Large weight matrices yield similar performance
  - Cubic GEMM scaling VS quadratic elementwise scaling

Multilayer Perceptron (MLP)



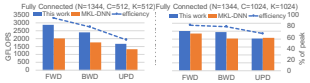
- An input layer $x \in \mathbb{R}^C$ is mapped to an output layer $y \in \mathbb{R}^K$ via: $y = W \cdot x$, where $W \in \mathbb{R}^{K \times C}$ is the weight tensor of the connections between the neurons.
- During the training process, N multiple inputs (N being mini-batch size) are grouped together yielding: $Y = W \cdot X$ with $W \in \mathbb{R}^{K \times C}$, $X \in \mathbb{R}^{C \times N}$ and $Y \in \mathbb{R}^{K \times N}$.
- After the output tensor Y is computed, a non-linear activation function $g()$ is applied on it.

Algorithm 2 Forward pass of Fully Connected Layer

Inputs: Weight $W[K_b][C_b][b_c][b_k]$, Input $X[N_b][C_b][b_n][b_c]$
Outputs: Output $Y[N_b][K_b][b_n][b_k]$

1: Based on $thread\_id$ calculate K_{b}_start, K_b_end, N_{b}_start and N_{b}_end to assign output work items
2: **for** $ib_n = N_{b}\_start \ldots N_b\_end$ **do**
3: **for** $ib_k = K_{b}\_start \ldots K_b\_end$ **do**
4: ▷ *Prepare batch-reduce GEMM arguments*
5: **for** $ib_c = 0 \ldots C_b - 1$ **do**
6: $A_{ptrs}[ib_c] = \&W[ib_k][ib_c][0][0]$
7: $B_{ptrs}[ib_c] = \&X[ib_n][ib_c][0][0]$
8: $Out = \&Y[ib_n][ib_k][0][0]$
9: **batchreduce_gemm**$(A_{ptrs}, B_{ptrs}, Out, C_b)$
10: $Y[ib_n][ib_k][0][0] \leftarrow g(Y[ib_n][ib_k][0][0])$

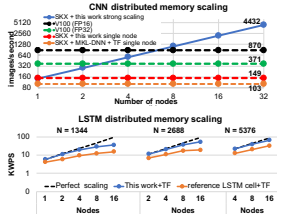
- We implement the multiplication $Y = W \cdot X$ via batch-reduce GEMM calls
- Apply the non-linear activation function $g()$ on the batch-reduce GEMM outputs once they are still hot in cache.
- Blocked layout to exploit locality & avoid conflict misses



- On Xeon 8180, the primitive based on batch-reduce GEMM outperforms MKL-DNN by 1.1× -1.3×.

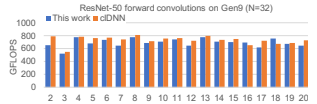
Distributed memory results

- 32-node cluster (Intel Omnipath interconnect), each node having two Skylake-SP (SKX) 8180 processors
- ResNet-50 training: integrated our CNN kernels into the GxM framework which scales efficiently on CPU clusters
- GNMT training: We replaced the Tensorflow LSTM cell implementation with our optimized LSTM cell that leverages the batch-reduce GEMM kernel.
- Exceeded the performance of vendor-optimized implementations by up to 2.3×



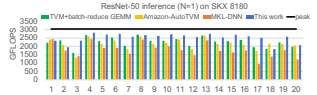
Performance portability on integrated GPU

- We developed the batch-reduce GEMM kernel in OpenCL.
- We implemented PoC forward propagation CNN kernels targeting Intel's integrated GPU Gen9 (Core i7 6770HQ) by leveraging the batch-reduce GEMM kernel.
- Our approach is within 3% of the vendor-optimized, ad hoc implementation (clDNN).



PoC integration within TVM

- We provide to TVM the forward propagation loop recipes in high-level Python code, and at the innermost loop nest level we invoke our batch-reduce GEMM kernel.
- Benchmarked inference use-case, ResNet-50 forward kernels (mini-batch N=1) on the SKX 8180 platform
- Our TVM implementation is 2% faster than the Amazon-AutoTVM *auto-tuned code* and 1.24× faster than the vendor optimized MKL-DNN library.



Summary and Conclusions

On this poster we presented the following work:

- we showed how the most popular DL algorithms (RNN/LSTM, CNN and MLP) can be formulated with batch-reduce GEMM as basic building block
- we demonstrated that our methodology outperforms vendor-optimized, low-level DL primitives by factors up to 1.4×
- we integrated our DL kernels into distributed frameworks, and optimized end-to-end workflows for GNMT and ResNet-50 training: we exceeded the performance of vendor-optimized implementations by up to 2.3×
- we highlighted the architectural-agnostic aspect of our methodology by matching the CNN kernel performance of a vendor-provided library on integrated GPUs
- we prototyped CNN kernels in a tensor compiler framework by harnessing our batch-reduce GEMM kernel, and matched the performance of *auto-tuned* inference TVM primitives

We upstreamed all changes to LIBXSMM and they can be found here: `https://github.com/hfp/libxsmm`. As future work, we plan to extend our DL primitives for a wider set of architectures/workloads, and also we intend to experiment with Tensor compilers' automatic polyhedral optimization, e.g. PlaidML.

For references of this work, please refer to the extended poster abstract and/or our extensive technical report: `https://arxiv.org/abs/1906.06440`.