MINI-BATCH SERIALIZATION: CNN TRAINING WITH INTER-LAYER DATA REUSE

Anonymous Authors[1]

ABSTRACT

Training convolutional neural networks (CNNs) requires intense computations and high memory bandwidth. We find that bandwidth today is over-provisioned because most memory accesses in CNN training can be eliminated by rearranging computation to better utilize on-chip buffers and avoid traffic resulting from large per-layer memory footprints. We introduce the MBS CNN training approach that significantly reduces memory traffic by partially serializing mini-batch processing across groups of layers. This optimizes reuse within on-chip buffers and balances both intra-layer and inter-layer reuse. We also introduce the *WaveCore* CNN training accelerator that effectively trains CNNs in the MBS approach with high functional-unit utilization. Combined, WaveCore and MBS reduce DRAM traffic by 73%, improve performance by 45%, and save 24% system energy for modern deep CNN training compared to conventional training mechanisms and accelerators.

1 INTRODUCTION

Convolutional neural networks (CNNs) are the state of the art for various vision applications. Training CNNs requires hundreds of thousands of compute- and data-intensive iterations. We observe that CNN training on current systems requires 3–4 times more off-chip memory bandwidth than necessary, reducing performance and wasting energy. We present a new training mechanism that significantly reduces bandwidth demands for the same arithmetic performance by better exploiting locality. We then develop a complementary accelerator that dramatically lowers training cost and time.

Conventional CNN training propagates data in lockstep across network layers for an entire mini-batch (typically 32–512 samples per processor (Szegedy et al., 2015; 2017; He et al., 2016)). Large mini-batches have per-layer memory footprints that exceed typical on-chip buffer capacity, resulting in high off-chip memory traffic (Fig. 1a). Directly applying locality techniques used in CNN inference (Parashar et al., 2017; Alwani et al., 2016) to training is ineffective because such techniques do not optimize locality across large mini-batches, and their design is not compatible with feature normalization (Ioffe & Szegedy, 2015).

Our *mini-batch serialization* (MBS) approach reduces memory traffic specifically for CNN training and exploits data reuse across layers (inter-layer data)—a first for training and for modern networks that include multi-branch modules and normalization layers. As illustrated in Fig. 1b, MBS breaks a mini-batch into *sub-batches* to reduce the per-layer memory footprint such that the inter-layer data of an entire sub-batch fits in on-chip buffers. MBS uses a different number of samples per sub-batch (sub-batch size) for different layers because pooling and strided convolution layers decrease the size of each feature and, hence, the total volume of features.

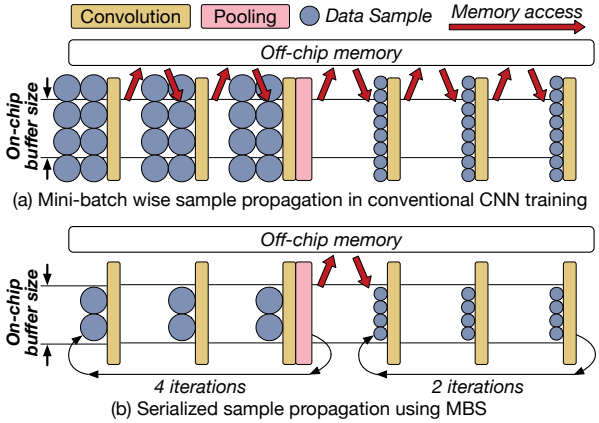


Fig. 1. A toy CNN architecture. MBS restricts per-layer memory footprint size smaller than the on-chip buffer.

MBS forms groups of layers such that each group has the same sub-batch size. Each sub-batch is then propagated with most inter-layer data staying on chip across the layers of a group; data needed during back propagation is stored off chip as well. Otherwise, layer output data is only written and later read from main memory between groups. MBS op-

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Systems and Machine Learning (SysML) Conference. Do not distribute.

timizes sub-batch sizes and layer grouping to balance data reuse between layers with reuse of parameters (weights) within a layer—weights are re-read for every sub-batch. MBS cuts memory traffic by 4× compared to conventional layer-by-layer mini-batch training across a number of popular deep CNNs.

While MBS optimizes locality and reduces memory traffic, applying MBS to modern deep CNNs and using it in a training accelerator introduces two challenges. The first is normalization as sub-batches prohibit the use of batch normalization. We therefore adapt group normalization (Wu & He, 2018) for use in the MBS flow and demonstrate the effectiveness of MBS training.

The second challenge is that MBS reduces per-layer parallelism, which potentially lowers the utilization of arithmetic units in the accelerator. We address this issue in two ways. First, we use the *im2col* (image-to-column) convolution algorithm that is commonly used by GPUs. It casts a convolution operation as a general matrix multiply (GEMM) (Chetlur et al., 2014). With im2col, the reduced parallelism from small sub-batches is effectively compensated for by the size of other feature dimensions. Second, we modify a traditional systolic array processing core, as used by some commercial accelerators (Jouppi et al., 2017; Lu et al., 2017), to better execute the tall and skinny GEMMs needed for im2col.

GEMM is blocked to utilize a systolic array, but idle time exists between the execution of any two blocks in a conventional design. To avoid this idle time, we augment each processing element with one additional 16b register that double buffers inputs and eliminates gaps between blocks. Our *WaveCore* accelerator with MBS achieves compute-unit utilization that is within 3% that of conventional training.

We use three recent deep CNNs to evaluate MBS and WaveCore: ResNet (He et al., 2016), Inception v3 (Szegedy et al., 2015), and Inception v4 (Szegedy et al., 2017). We show that MBS saves DRAM accesses by 74% 72%, 74%, improves training performance by 50%, 39%, 41%, and saves 26%, 24%, and 24% energy for ResNet50 and Inception v3 and v4, respectively. We also demonstrate that MBS enables high-performance training accelerators that use much more affordable but slower off-chip memory (e.g., LPDDR)—even with 60% less memory bandwidth, training performance is still 16% above the baseline design.

We summarize our contributions below:

- We introduce Mini-Batch Serialization (MBS), a hardware-resource aware CNN training optimization that significantly reduces off-chip memory traffic and can thus accelerate CNN training and reduce training cost. MBS balances intra- and inter-layer locality and cuts DRAM traffic by 3.8× for modern CNNs.
- We show how MBS exploits locality within multi-branch modules to achieve the 3.8× traffic reduction (traffic is reduced by 3.3× without this multi-branch optimization).
- We augment a conventional systolic array architecture and optimize it to effectively accelerate MBS-based training. Our WaveCore accelerator maintains high processing-element utilization for modern CNNs and utilizes MBS to provide high performance with both high-bandwidth HBM2 DRAM (as used by GPUs and Google's TPU v2 and v3) and even lower-cost and higher-capacity GDDR5 and LPDDR4 DRAM systems.

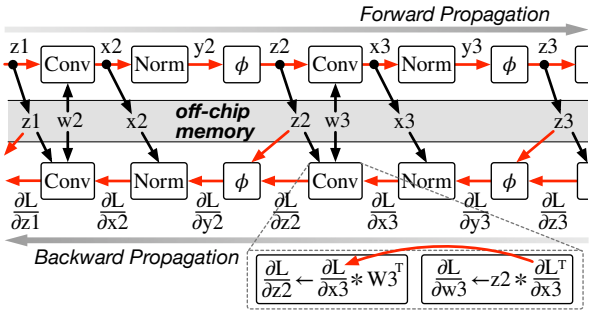


Fig. 2. Dataflow in forward and backward propagations. Red arrows show the reusable data between layers.

2 Data Locality in CNN Training

CNN training consists of forward and back propagation phases. Fig. 2 illustrates the major data elements needed for training and their reuse patterns with red arrows indicating opportunities for on-chip buffers to reduce memory bandwidth requirements and black arrows indicating accesses to main memory. In both phases there is direct producer-consumer locality between layers—*inter-layer data* that can be buffered if it is not too large. The outputs of convolution, normalization, and activation layers in forward propagation (x, y, and z in the figure) are immediately used by their following layers. Normalization layers exhibit additional reuse because they iterate over inputs to first compute the mean and variance before normalizing the data (Ioffe & Szegedy, 2015). The convolution outputs and the activations are stored in off-chip memory for reuse in back propagation, because their large storage requirements and long data reuse distance prevent on-chip buffering.

Back propagation exhibits even greater potential for inter-layer reuse. The loss gradients (with respect to x) are reused twice by a convolution layer to compute the gradients of weights and loss (with respect to z). Also, the convolution output stored in memory is reused multiple times to compute the gradients of the normalization layer parameters and the loss gradients (with respect to x). Activations read from memory are also used twice: z is used for convolution gradients and the derivative of z for activation gradients.

The Problem with CNN Training Memory Footprint.

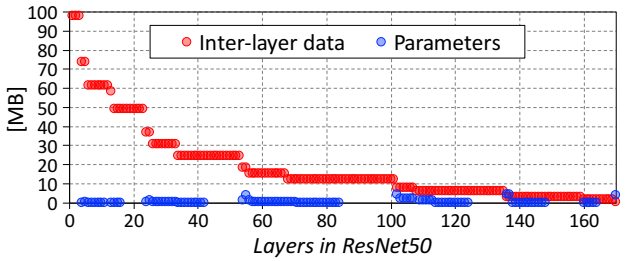


Fig. 3. The size of inter-layer data and parameters of each layer in ResNet50.

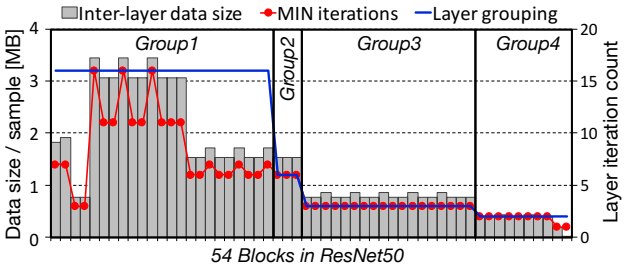


Fig. 4. Per-block inter-layer data size, required layer iterations, and MBS layer grouping for ResNet50 with 32 samples.

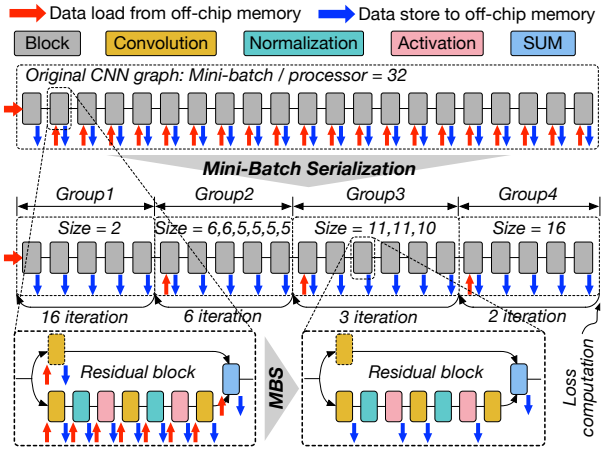


Fig. 5. Baseline and MBS ResNet training flow (the number of blocks is illustrative only).

CNN training is typically done with mini-batches of 32–512 samples (possibly distributed across multiple processors) (Sutskever et al., 2013). Larger mini-batches reduce model parameter update frequency and training iterations, thus reducing training time and energy (Li et al., 2014; Goyal et al., 2017). An additional benefit is that larger data parallelism can be used to maintain high compute unit utilization and help in distributing work across multiple processors (Das et al., 2016).

However, a larger per-processor mini-batch with many samples increases the memory footprint of each layer, limiting the opportunity to reuse data on chip. Fig. 3 shows the per-layer footprint of ResNet50 with a mini-batch size of 32 and a word size of 16b (in the forward phase). Only 9.7% of inter-layer data can be reused even with 10MiB on-chip storage, leading to very significant memory bandwidth waste for storing and refetching data. This problem is even more severe for larger mini-batch sizes, which are desirable as per-processor arithmetic performance and main memory capacity improve.

3 Mini-Batch Serialization

The primary goal of MBS is to improve reuse by exploiting inter-layer data locality. The key to MBS is partially serializing a mini-batch (propagating a small sub-set of a mini-batch at a time) to control per-layer memory footprint without impacting training accuracy. MBS is based on our insight that if the data synchronization points for functional correctness are maintained and an appropriate normalization algorithm is adapted, even processing a single sample at a time through all network layers does not alter the training result. The trivial serialization of one sample at a time, however, has two crucial drawbacks.

First, while baseline training reads weights and writes weight gradients just once per layer, full serialization re-reads weights and partial gradient sums for each sample and updates the partial sums once per sample as well. Second, data parallelism within a single sample can be limited in some layers, degrading resource utilization and performance (especially when mapping to a highly-efficient systolic architecture).

An improvement on full serialization is to process multiple samples at a time (a *sub-batch*) to provide some intra-layer weight reuse and extra parallelism, as long as the footprint at any point in the sub-batch does not exceed the on-chip buffer capacity. The entire mini-batch is then processed in several sub-batch iterations. However, the footprints of early layers are large and only a small sub-batch can be formed (1–2 samples), limiting the benefits of this approach.

MBS goes much further and balances locality of intra-layer weight reuse and parallelism with inter-layer locality. We do this by varying the number of samples per sub-batch across layers such that layers that can support more samples require fewer iterations and can benefit from the greater parallelism and locality. This is possible because pooling and strided convolution layers decrease feature size and volume for deeper layers.

Layer Grouping Optimizes Reuse. Optimizing layer groups balances intra- and inter-layer locality tradeoffs. The MBS algorithm forms initial layer groups by grouping adjacent layers that require the same number of sub-batch iterations. This is shown in Fig. 4 where grey vertical bars represent the data volume required for the inter-layer data per layer (or one multi-branch module *block*) of ResNet50, and the red line represents the resulting minimal sub-batch iteration count for each layer. Then layer groups are merged

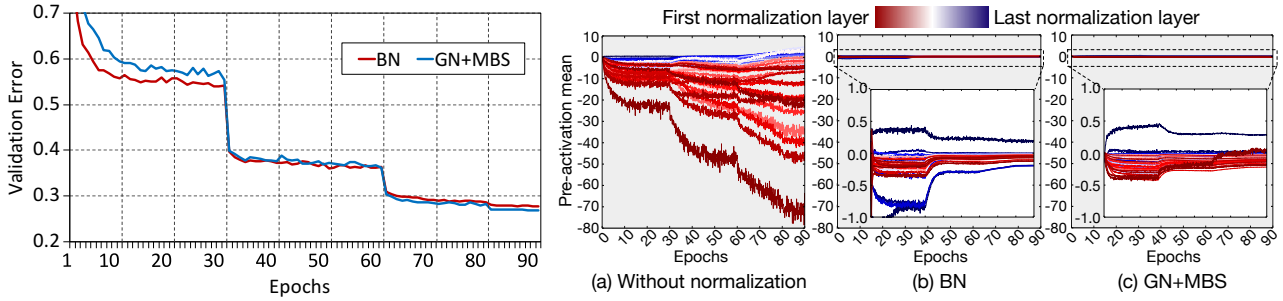


Fig. 6. ResNet50 training with GN with MBS and BN: validation error (left) and pre-activation mean of each normalization layer with BN and GN zoomed (right). Mini-batch size of 64, an initial learning rate 0.05, and learning rate decays of 0.1 at epochs 30, 60, and 80.

to improve overall locality: Groups are merged by reducing the sub-batch size of one group to that of an adjacent group. The first group then requires more iterations (with more weight and gradient accesses), but inter-layer reuse increases across the two layers where the groups meet. The resulting grouping for this optimization for ResNet50 is shown with the blue line in Fig. 4.[1]

The mini-batch is then processed in several sub-batch iterations ($\lceil \frac{mini-batch\ size}{sub-batch\ size} \rceil$) within each group as shown in Fig. 5, which emphasizes how locality is increased and memory traffic reduced across features and weights.

Data Reuse Within Multi-Branch Modules. Fig. 5 also shows how MBS applies the same sub-batch approach to a multi-branch residual module of ResNet50. Such multi-branch modules are common in CNN architectures and offer additional reuse opportunities. Both the main path and shortcut branch share an input, and when they merge, their outputs are summed. Therefore, the module inputs should stay on chip until both paths have consumed them, and the output of the shortcut branch should stay on-chip while the main path output is computed. MBS does this by provisioning buffer space based on the needs of multi-branch *blocks*, where a block includes all the branches that share split and merge points—MBS essentially treats such a block as a layer for optimizing locality.

Maintaining locality for such shared nodes leads to additional storage requirements. The per-sample size is calculated by Eq. 1 where: D_{in} and D_{out} indicate the sizes of the main-branch input and output; $D_{shortcut}$ is the size of the shortcut path output; L is the number of layers in the main branch; and b and l represent a specific branch and layer.

$$\frac{Space}{Sample} = D_{cond}(l) + \max_{1\le b\le 2,\ 1\le l\le L} D_{in}(b,l) + D_{out}(b,l)$$
$$D_{cond}(l) = (b\neq 1) D_{out}(1,1) \quad (1)$$

Similarly, for Inception modules (Szegedy et al., 2015; 2017), the block input is reused between branches, and the concatenated block output is eventually reused in the following layer. Therefore, MBS keeps both the block input and output on chip while executing the branches. The space required is shown in Eq. 2, where B indicates the number of branches in a module and other notation is as above.

$$\frac{Space}{Sample} = \max_{1\le b\le B,\ 1\le l\le L} D_{in}(b,l) + D_{out}(b,l) + D_{cond}(l)$$
$$D_{cond}(l) = (l\neq 1) D_{block\_in} + (l\neq L) D_{block\_out} \quad (2)$$

Back Propagation. In back propagation, MBS optimizes locality for both newly computed results and for data reloaded from the forward path. For example, as shown in Fig. 2, MBS reuses the reloaded gradients more than once. Furthermore, both convolution and ReLU layers use activations from the forward path. However, only the gradient of ReLU is needed, which is always exactly 0 (for negative activations) and exactly 1 (for positive); thus, MBS uses a single bit per ReLU gradient instead of a 16b number. We also allocate buffer space for normalization layers to reuse their inputs to compute their gradient and loss. As in the forward pass, reuse in back propagation is made possible by MBS processing one sub-batch at a time.

Data Synchronization. MBS maintains the original synchronization points across the entire mini-batch. Therefore, MBS accumulates the partial gradients of all learning parameters across all sub-batches. This requires storing partial results to memory, which is not needed in the conventional flow. However, this overhead is dwarfed by the improved reuse of layer outputs, especially considering that deeper layers with large weights are iterated over only a few times.

3.1 Feature Normalization in MBS

While batch normalization (BN) is widely used in many modern CNNs, it is incompatible with MBS because BN requires many samples to work well and improve accuracy (Ioffe & Szegedy, 2015)—MBS cannot serialize computation if data across an entire mini-batch (per processor) is needed for normalization. Instead of using BN, we adapt group normalization (GN) (Wu & He, 2018) to MBS. GN normalizes across features within a subset of channels in a

[1] We also experimented with an optimal grouping of layers using exhaustive search, which improved traffic and performance by roughly 1% compared to our greedy optimization.

single sample, as opposed to across an entire per-processor mini-batch. Thus, GN can be made compatible with MBS.

To use GN with MBS, the per-channel GN scale and shift parameters must be re-fetched at every sub-batch iteration within a layer group. Additionally in backward propagation, the gradients of these parameters must be accumulated across all sub-batches just like the weights of convolution layers. However, since the size of these parameters is only two times the number of channels per layer, they can easily be stored in the on-chip buffer and incur no overhead.

We confirm previous results and demonstrate that both GN and BN provide comparable training effectiveness. Fig. 6 compares the validation error curves with BN and MBS-GN when training ResNet50 on ImageNet (Deng et al., 2009). Fig. 6 also shows that both MBS-GN and BN provide similar normalization, in that both have similar pre-activation (output of normalization) distributions across layers (unlike training without normalization).

4 WaveCore Accelerator

We explain the WaveCore accelerator operation in two parts. First, we discuss the core compute engine of WaveCore and any modifications we make to adapt the conventional systolic array for MBS. Second, we describe the overall accelerator architecture, which includes additional components for processing normalization, activation, and pooling layers. We also estimate the area and power of WaveCore.

4.1 Systolic Array Core

WaveCore uses a large systolic array as its main compute unit. A systolic array is (typically) a two-dimensional mesh of many simple and efficient processing elements (PEs). At each cycle of a kernel, each PE applies the same computation to its inputs and then passes the computed result or its unmodified inputs to one or more of its neighbors. All PEs communicate only with adjacent PEs such that there is minimal data movement and high computational concurrency (Kung, 1982). Computation consists of pipelining inputs from the top and left (for example) edges of the array and obtaining results at the bottom. The large compute throughput required for convolutional and fully-connected layers, along with the repetitive computation and large data reuse are a good match for a systolic array, as found in Google's TPU ML accelerators (Jouppi et al., 2017; Dean, 2017). Like prior work, our proposed PE has mixed precision units: 16b inputs are multiplied with accumulation performed in 32 bits to reduce both computation and data traffic overheads (Micikevicius et al., 2017). Also like prior work (Dean, 2017), we use a 128×128 systolic array for high performance and to circumvent power delivery challenges.

A systolic computation is often divided into multiple *waves*,

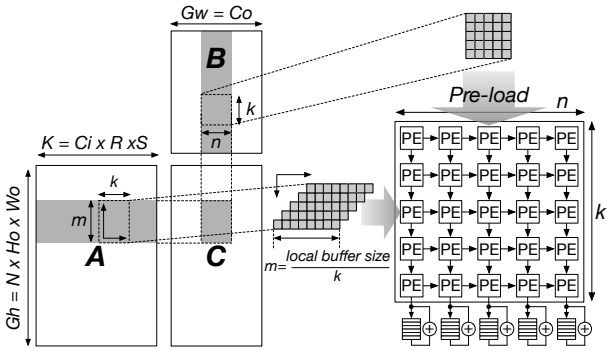


Fig. 7. GEMM dimensions , tiling, and mapping of each tile to the systolic array of a convolution layer in forward propagation.

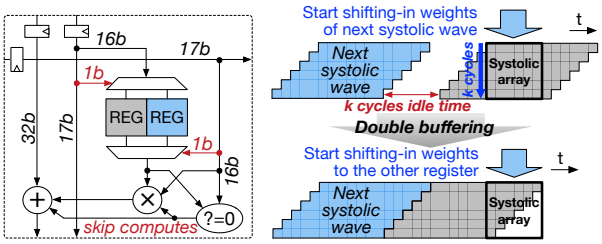
Tab. 1. GEMM matrix dimensions for *im2col* convolution for different CNN training phases. N, C, H, W indicate the sub-batch size, the channel count, and the height/width of each feature (i and o denote input and output features, respectively), and R, S are the height and width of each filter.

Convolution Phase	G_h dimension	G_w dimension	K dimension
Forward	$N \times H_o \times W_o$	$C_i \times R \times S$	C_o
Backward Loss	$N \times H_i \times W_i$	C_i	$C_o \times R \times S$
Backward Weight	$C_i \times R \times S$	C_o	$N \times H_o \times W_o$

where each wave proceeds with inputs flowing toward outputs without any stalls or changes to the computational pattern. Between waves, it is sometimes necessary to let the pipeline through the array drain and then refill. This introduces idle time which reduces utilization and hence hurts performance and efficiency. Convolution and matrix operations have efficient systolic implementations that have little idle time if an entire mini-batch is processed together. However, MBS processes an often small sub-batch, which significantly reduces the utilization and performance of a conventional systolic array design. We address this challenge and maintain high systolic array utilization for MBS using a combination of two techniques.

Maintaining High Compute Unit Utilization with im2col. First, instead of directly mapping a convolution computation to a systolic array, we use a method that transforms a convolution into a matrix multiplication. We do this because efficient direct convolution on a systolic array requires tuning for every possible sub-batch size, which is difficult to do with the MBS approach which optimizes groupings to arbitrary size. We use the *im2col* (image-to-column) general matrix-matrix multiplication (GEMM) algorithm for convolution, which is commonly used in GPU-accelerated kernels (Chetlur et al., 2014). Convolution with *im2col* rearranges the address pointers to the convolution inputs in a way that is straightforward to feed into a systolic array. The GEMM dimensions (G_h, G_w, K) are determined by the convolution configurations as summarized in Tab. 1.

If the size of G_h, G_w, or K is smaller than the systolic array size, the compute units are significantly underutilized. How-



(a) PE design: Data double buffering and selection

(b) Weight shift-in start time change by stationary data double buffering

Fig. 8. Removing inter-wave idle time by weight double buffering and control signal shift.

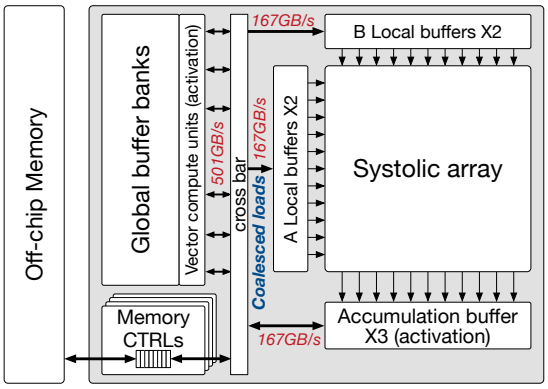


Fig. 9. Per-core architecture of the WaveCore accelerator.

ever, with the networks we evaluate, this does not become a significant limitation: Early layers with small sub-batches have large features and while the feature sizes of later convolution layers are small, their large sub-batch size compensates (shown by the red-colored dimensions in Tab. 1).

Systolic Dataflow for im2col GEMM. We block this im2col GEMM into multiple $m \times n$ tiles, which are processed in sequence through the array. Each tile corresponds to a portion of the output matrix (C). The width of each tile is equal to the width of the systolic array (n). The height (m) is chosen to maximize the size of a tile, thus minimizing the number of tiles per layer and improving utilization: $m = \frac{Local\ buffer\ size}{k{=}systolic\ array\ height}$. This is illustrated in Fig. 7.

Each tile is processed using multiple waves through the systolic array, where each wave multiplies a block of input matrix A by a block of input matrix B. A block from B is first read one row at a time. Each row is shifted down until the array has one element of B per PE (this takes 5 cycles in the toy example of Fig. 7). Then, a block of A is pipelined into the array with results for each element of C eventually accumulated at the bottom of the array as shown in the right side of Fig. 7. Notice that in the figure, cycle 6 corresponds to the first row of the block of A having been multiplied and then accumulated by the first column of the block of B. In the following cycle, the second row of the A block completes its pipeline through the first column, while the first row of A now completes its dot product with the second column of the B block (and its output is at the bottom of the second column of the systolic array).

Once a wave as described above completes, the next blocks of A and B are processed. As additional blocks are processed, their outputs are added to the current values of the C tile (a reduction across waves), eventually completing a tile of C in $\lceil K/k \rceil$ waves (K is the dimension of the input matrices and k is the PE array height).

Gap-less Waves with Weight Double Buffering. The flow described above has one significant problem. Before every multiplication of blocks of A and B, the B block is read and distributed to the PEs, which requires k (PE array height) cycles (for reading and inter-PE shifting). No arithmetic occurs during these k cycles, which decreases performance (upper half of Fig. 8b).

To remove this inter-wave idle time, we modify the basic PE design to double buffer weights (Fig. 8a)—the next wave's weights are fetched and distributed into a second register within each PE while the current wave is still being processed. As the current wave starts draining from the PE array, the following wave starts immediately by feeding in a new block of A and multiplying by the second register that stores the next set of weights from B. Thus, there are no gaps between waves and an entire tile of C is computed without any idle time beyond the initial fill and final drain of the pipeline. In addition to the extra register in each PE, a minor further change is that a select signal for choosing which weight register to use is propagated along with the inputs of A and B. This optimization significantly boosts performance at very low cost: the simple 1b local signal between every two PEs and a 16b register and multiplexer between the two registers per PE. As in prior work, we also check for zero inputs and skip arithmetic in such cases to reduce energy consumption (Parashar et al., 2017).

4.2 Overall Processor Architecture

In addition to the systolic cores, the WaveCore CNN training accelerator contains several more structures and units. Fig. 9 illustrates the overall architecture of one core of the processor. There are two such cores in our proposed design that are connected by an on-chip network, similar to TPU v2 (Dean, 2017). We describe these structures and estimate the area and power requirements of WaveCore below.

Local Buffers. Both A and B local input buffers are double-buffered. Double buffering enables the overlap of computation within the PEs with accesses to the global buffer and to memory and allows for very simple coarse-grain control of data transfers between buffers and memory. We choose the minimal size for each buffer, such that PEs never directly access the global buffer or memory, as this avoids access-related stalls. A half-buffer of B stores a 16b word for each

PE and is thus 32KiB ($128\times128\times16b$). Each half-buffer for A is 64KiB because A blocks need to be twice as large as B blocks to avoid inter-wave idle time. The output accumulation buffer is triple-buffered because it holds the current output tile while the previous tile is being written to memory and the partial gradient sums for the next tile are read. Each part of this buffer holds an entire tile of C and is 128KiB. Note that while outputs are summed in 32b precision, the final write to the output buffer quantizes to 16b precision.

Global Buffers. The baseline global buffer is 10MiB and has 16 banks. This is sufficient for using MBS with modern CNNs and avoiding bank access conflicts. The global buffer is connected to all local buffers via a crossbar. To avoid duplicated data loads from the global buffer, we have memory load coalescing units that maintain high effective bus bandwidth utilization. Our processor operates at a 0.7GHz clock frequency, and the data bandwidth of local and global buffers are set to fully support the systolic wave pipelining.

Main Memory. The off-chip memory is connected to memory controllers, which communicate with the on-chip buffers via the crossbar switches. Our baseline WaveCore uses a single HBM2 stack with 4 dice (Joi, 2016), which provides 8GiB off-chip DRAM with 300GiB/s data bandwidth over 8 channels (4 channels per core). We choose HBM2 because it is used by other modern training accelerators (Dean, 2017; nvidia, 2017). We later show that cheaper GDDR or even LPDDR memory can be sufficient for WaveCore.

Vector and Scalar Computing Units. The systolic array is used for convolutions and fully-connected matrix operations, but cannot be efficiently utilized by normalization, pooling, and activation layers, which require a relatively small number of arithmetic operations. Such layers are memory bandwidth bound, and we therefore process them using scalar and simple vector units that are placed close to the global buffer where their outputs are then stored.

Scalability. We describe and evaluate WaveCore with two cores, but compute throughput can be easily scaled with larger mini-batches distributed across multiple accelerators or additional cores. As each accelerator or core conducts the same job, we can use MBS within each WaveCore and only communicate for loss computation and parameter reduction and update.

Area Estimation. We estimate the die area of WaveCore at 45nm technology and scale this estimate to 32nm to compare with other deep learning accelerators Tab. 2. The estimated total area of the two-core WaveCore is 534.0 mm^2. We use a 24T flipflop design as reported in (Kim et al., 2014) and the floating point multiplier and adder designs reported in (Hickmann et al., 2007). Each PE requires 12,173 um^2 and both multiplier and adder take more than 90% of the PE area. The estimated area of the 128×128 PE array is 199.45 mm^2, which accounts for 67% of WaveCore's area. The size of the global buffer and the vector compute units per core are estimated at 18.65 mm^2 and 4.33 mm^2, respectively. The crossbar has 24 256b-wide ports (32B memory access granularity). The area occupied by the network and the crossbar expands the chip width by 0.4mm, following the approach used to evaluate Dadiannao (Chen et al., 2014b).

Tab. 2. Accelerator specification and comparison.

	GV100	TPU v1	TPU v2	WaveCore
Technology (nm)	12 FFN	28	N/A	32
Die Area (mm^2)	812	$\leq$ 331	N/A	534.0
Clock Freq (GHz)	1.53	0.7	0.7	0.7
TOPS / Die	125 (FP16)	92 (INT8)	45 (FP16)	45 (FP16)
Peak Power (W)	250	43	N/A	56
On-chip buffers (MiB)	33[1]	24	N/A	20 (2×10)

[1] Sum of L2, shared memory, and registers

Tab. 3. Evaluation configuration description.

Configuration	Description
Baseline	2-level GEMM blocking
ArchOpt	Baseline + weight double buffering
IL	ArchOpt + inter-layer data reuse
MBS-FS	IL + serialize all layers using the same sub-batch size
MBS1	IL + greedy layer grouping
MBS2	MBS1 + inter-branch data reuse

Tab. 4. Off-chip memory configuration.

Memory type	Per-chip configuration	Chip #	Total BW
HBM2	300 GiB/s, 8 GiB, 8 channels	x1	300 GiB/s
HBM2×2		x2	600 GiB/s
GDDR5	32 GiB/s, 1GiB, 1 channel	x12	384 GiB/s
LPDDR4	29.9 GiB/s, 2GiB, 1 channel	x8	239.2 GiB/s

Power Modeling. We use a convolution layer that exhibits 100% systolic-array utilization to estimate the peak power consumption of WaveCore. WaveCore operates at 0.7GHz, which is $< 1/2$ compared to GV100 (1.53 GHz) and the same as TPU v2 (Dean, 2017). WaveCore consumes a maximum of $56W$ (Tab. 2). Here, we use a HBM2 as the off-chip memory and model its power using the Rambus power model (Vogelsang, 2010) in 22nm technology. The SRAM buffer power is calculated with CACTI (Chen et al., 2012) configured for 32nm. The power consumed by multipliers and adders is taken from (Han et al., 2016) and filpflops from (Fuketa et al., 2013). The link and router power is calculated with Orion2.0 (Kahng et al., 2009).

5 Evaluation Methodology

We evaluate the locality benefits of MBS and the performance and energy of WaveCore on three well-known modern *deep CNNs*: ResNet (He et al., 2016), Inception v3 (Szegedy et al., 2015), and Inception v4 (Szegedy et al., 2017). We also evaluate a shallower CNN (AlexNet (Krizhevsky et al., 2012)) with few memory BW bound layers such as normalization and pooling. We use mini-batches of 32 samples per core (64 per chip) for the deep CNNs and 64 samples per core for AlexNet because of its smaller training context. We use 16b floating point for all

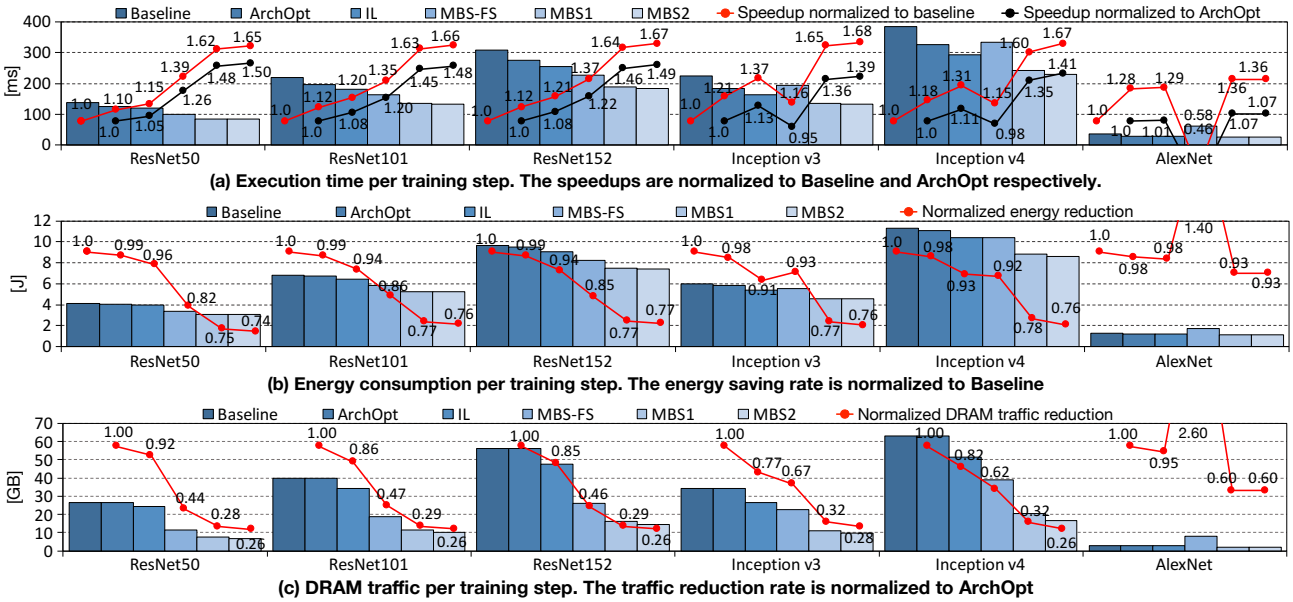


Fig. 10. DRAM traffic, performance and energy consumption sensitivity to the proposed network architecture reconfiguration and HW architecture optimization methods.

CNNs with mixed-precision arithmetic (16b multiplication and 32b accumulation) (Micikevicius et al., 2017).

For each network, we evaluate several execution configurations as summarized in Tab. 3: **Baseline** uses two-level GEMM input matrix blocking for effective data reuse within each convolution and FC layer (Kurzak et al., 2012); **ArchOpt** adds weight double buffering for better PE utilization (all other configurations use ArchOpt), **Inter-Layer (IL)** reuses the shared data between layers but only when the per-layer memory footprint of the entire mini-batch fits within the on-chip buffer (i.e., not using the MBS approach), **MBS-FS** is naive MBS that fully serializes a mini-batch such that all layers in the CNN have the same sub-batch size, **MBS1** greedily forms layer groups to simultaneously optimize both intra- and inter-layer data reuse, and **MBS2** additionally reuses the inter-branch data which requires different layer grouping than MBS1. We compare WaveCore with MBS to an NVIDIA GV100 running Caffe (Jia et al., 2014) and report values averaged over 10 GPU iterations.

The WaveCore simulator accounts for all memory, buffers, and on-chip interconnect traffic as well as the arithmetic operations. The default WaveCore uses a single HBM2 chip with 4Hi stacks. We also scale memory bandwidth up to two HBM2 stacks to launch a larger mini-batch per accelerator (and to more closely match commercial accelerators). Because MBS significantly reduces memory traffic, we also evaluate lower-bandwidth main memory options that are cheaper and offer higher capacity (GDDR5 and LPDDR4). The off-chip memory configurations of WaveCore are listed in Tab. 4.

6 Evaluation Results

Fig. 10 compares the per-training-step execution time, energy consumption, and DRAM traffic of our proposed technique. In each of the subfigures, bars show absolute values and lines show relative ones. We normalize execution time separately to both Baseline and ArchOpt to isolate the impact of the architectural and algorithmic contributions of WaveCore and MBS.

Compared to Baseline, ArchOpt improves performance by 10–28% across CNNs by removing the idle time between systolic waves. The gain is particularly large for AlexNet because AlexNet has mostly convolution layers with few memory-BW bound layers. Similarly, while not shown in the figure, ArchOpt provides more benefit with MBS because the large reduction in memory traffic increases the relative impact of idle compute time. For example, MBS2 without ArchOpt is only 17% faster than Baseline on average, whereas speedup is 68% for MBS2 with ArchOpt, a 43% improvement on average. ArchOpt has little energy benefit ($\sim$ 2%) because it conserves only static energy.

Inter-layer (IL), which is similar to prior locality approaches used for inference, has only a modest impact on performance, energy, and traffic because many layers have large footprints that exceed the buffer size.

MBS-FS, which uses a single sub-batch size (and thus a single group) substantially reduces DRAM traffic (33–56%) for the deep CNNs because it utilizes inter-layer locality well. However, with a small sub-batch size, the time needed for the extra reads and writes of weight gradients used to reduce weight gradients across sub-batches cannot be hidden,

which reduces performance. This is evident in the performance trends of Inception v3 and v4, where MBS-FS is worse than IL. AlexNet exhibits a much larger performance loss with MBS-FS because it has three FC layers with large weights and the extra weight reads increase main memory traffic by 2.6×.

MBS1 balances inter- and intra-layer reuse and achieves large improvements in performance (35–48%) and DRAM traffic (68–72%) for the deep CNN compared to ArchOpt. AlexNet shows smaller gains as it lacks memory-BW bound layers. MBS1 also shows 22–25% energy saving for the deep CNNs compared to Baseline by reducing the DRAM energy portion from 21.6% to 8.7%. As WaveCore skips multiplication and addition when one of the inputs to a PE is zero, the contribution of DRAM traffic reduction to the overall energy saving is high. It is important to note that global buffer traffic is increased by a similar amount as DRAM traffic is decreased. However, there is still a large net energy saving because a global buffer access energy is 8× lower than that of DRAM.

MBS2 reduces DRAM traffic by an additional 3–8% and improves training performance by up to 5% compared to MBS1. MBS2 needs additional global buffer space to store the data at the shared multi-branch nodes, so the number of sub-batch iterations is larger than with MBS1. While more iterations imply a larger overhead for re-reading weights and gradient, the traffic saved by the reuse between branches is greater. The gain is bigger for Inception v3 and v4 because the Inception modules have more branches and reuse opportunity scales linearly with the number of branches.

In summary, the highly-optimized MBS2 improves DRAM traffic by 72–74%, training performance by 39–50%, and energy consumption by 24–26% for the deep CNNs.

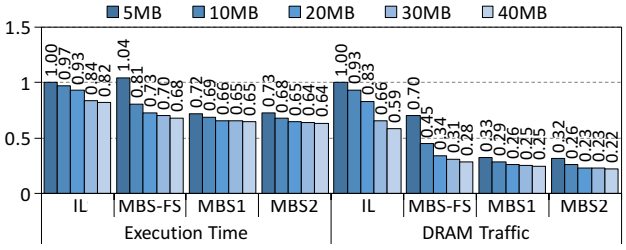


Fig. 11. Memory traffic and performance sensitivity of ResNet50 to the global buffer size (Normalized to IL with 5MiB).

Sensitivity to Global Buffer Size. Another benefit of MBS is its low sensitivity to on-chip storage capacity. To showcase this, we compare the execution time and DRAM traffic per training step of ResNet50 for different configurations with different global buffer sizes (Fig. 11). The per-core global buffer size is scaled from 5MiB to 40MiB and execution time and traffic are normalized to IL at 5MiB (ResNet's MBS scheduling requirement is smaller than 5MiB). Even

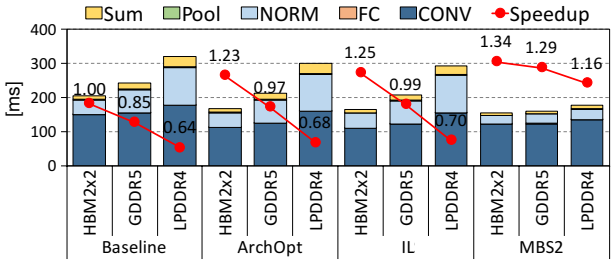


Fig. 12. ResNet50 training performance sensitivity to the memory type and the execution time breakdown by layer type.

with a 40MiB global buffer, only 41% of DRAM traffic is saved by IL; MBS1 and MBS2 save almost twice as much traffic even with a 5MiB buffer. IL with 40MiB also provides less performance benefit than both MBS1 and MBS2 at just 5MiB. Both MBS1 and MBS2 show little performance and DRAM traffic variation for different buffer sizes because they simultaneously balance both intra- and inter-layer reuse. In contrast to the optimized MBS1 and MBS2, MBS-FS again suffers from the impact of reading and writing gradient partial sums.

Sensitivity to DRAM BW. Fig. 12 highlights the ability of MBS to enable high performance even with lower-cost, lower-bandwidth memories. The figure compares the per-step training time of different configurations using various memory types (speedup is normalized to Baseline with 2×HBM2). The bandwidth of GDDR5 and LPDDR4 is 64% and 40% that of HBM2×2, respectively. While all implementations suffer from decreased bandwidth, the improved locality with MBS2 makes it far less sensitive with only a 5% performance drop when using off-package GDDR5 and a $< 20\%$ drop with low-cost LPDDR4. In this experiment, the off-chip memory space has been increased to 16GB to train 64 samples per core (128 per WaveCore) because off-package memories offer higher capacity.

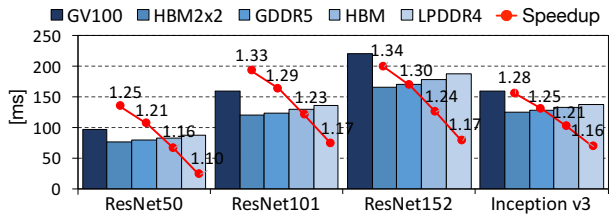


Fig. 13. NVIDIA GV100 GPU performance comparison to WaveCore + MBS2 with different memory types.

Comparison to GPU. Fig. 13 compares the measured execution time per training step of an NVIDIA GV100 GPU with our estimates for WaveCore with different DRAM configurations. Although a single WaveCore has 30% the peak compute and 27% the memory bandwidth (LPDDR4) of GV100, it still exhibits better training performance. The performance gap widens as the network depth increases because many layers with low data parallelism cannot effi-

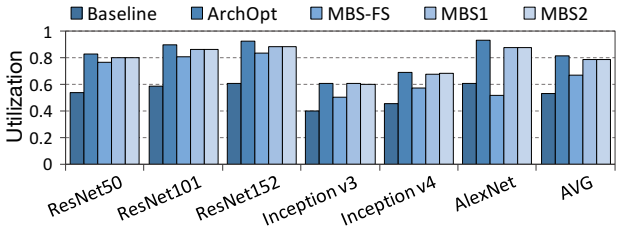


Fig. 14. Systolic array utilization of different CNNs.

ciently utilize the wide compute resources of the GV100.

Systolic Array Utilization. As MBS propagates only a fraction of the mini-batch for each sub-batch iteration, it is important to observe its impact on the systolic core utilization. Fig. 14 compares the utilization of convolution and FC layers for different CNNs. To isolate the impact of sub-batch size and the parallelism it makes available on utilization, this experiment uses unlimited DRAM bandwidth. Baseline suffers from low core utilization (average of 53.8%) due to the inter-wave idle time. Double buffering with ArchOpt increases the average utilization to 81.5%. MBS-FS exhibits lower utilization (66.7%) because the sub-batch size is determined solely by the large early layers. Optimizing reuse with different sub-batch sizes across layer groups with MBS1 and MBS2 regains the lost utilization and brings it up to 78.6%, within 3% of a full mini-batch. This small difference is largely a result of a few early layers with small channel counts, which result in particularly narrow tiles that do not fully utilize WaveCore's 128×128 systolic array. Later layers exhibit almost 100% utilization.

7 Related Work

To our knowledge, no prior work has addressed locality-optimizations for CNN training. Instead, we discuss methods proposed for inference accelerators. Most inference accelerators optimize CNN scheduling to better utilize intra-layer locality (Gao et al., 2017; Chen et al., 2017; Lu et al., 2017; Chen et al., 2014a; Du et al., 2015; Jouppi et al., 2017). They mainly focus on the data flow within a processing array, reducing data re-fetches by unrolling, or optimizing data access patterns within a convolution layer.

SCCN (Parashar et al., 2017) is a scheduling method and architecture that reuses inter-layer data in CNN inference. SCNN uses the on-chip buffer to hold both the input and output features of each layer along with all weights. This is possible with a reasonable on-chip buffer size because SCNN relies on the fact that inference uses a single sample (mini-batch of 1), that features between layers are sparse because of ReLU, and that weights are even more sparse because they are pruned once training is complete. Together, an entire network can fit within an on-chip buffer.

However, the SCNN approach cannot be used for training because the same conditions do not hold true: mini-batches are large resulting in layer outputs that exceed buffer size, convolution layer outputs are not sparse, and weights are not sparse before pruning (Han et al., 2015).

Fused-Layer CNN (Alwani et al., 2016) is an inference flow that also utilizes inter-layer data. The approach is to divide the initial input to the CNN (the input feature map) into tiles and propagate one tile through multiple layers. Each convolution layer uses its input tile to produce a smaller output tile (because output cannot be produced for bands along the tile edges). The overlap between tiles is exploited via dedicated caches. While effective for the networks evaluated in (Alwani et al., 2016), Fused-Layer CNN can not be applied to training modern deep CNNs, because: (1) convolution layers with small feature maps and large channel counts and weight data (deeper layers in modern CNNs) do not exhibit sufficient inter-tile locality; (2) normalization layers are incompatible with the tiling used for the depth-first propagation; (3) the inter-layer communication pattern in multi-branch modules, as well as in back propagation, is not only a direct communication between one layer to its following one; and (4) tiles shrink as they are propagated depth-first through the network, which limits available parallelism and likely hurts PE utilization.

8 Conclusion

We introduce MBS, a mechanism to reuse the inter-layer data in CNN training and balance its locality with that of intra-layer data. MBS reconfigures the CNN computation graph by partitioning a mini-batch of samples into sub-batches whose memory footprint fits within on-chip storage. We show that MBS reduces the volume of DRAM accesses by up to 74% while providing a high processing-element utilization of 79%. Additionally, we are the first to demonstrate and exploit data reuse opportunities between branches in CNN multi-branch Residual and Inception modules. To efficiently use MBS CNN training, we introduce WaveCore, a systolic-array based CNN training accelerator. We design WaveCore to double-buffer data within its processing elements to remove idle time between the systolic waves used to compute the convolution and fully-connected layer outputs. Our evaluation demonstrates that we expect single WaveCore with MBS to achieve higher performance than one GV100 GPU despite having the GPU having 3× higher peak performance and memory bandwidth.

Furthermore, the high locality MBS achieved by balancing intra- and inter-layer reuse makes WaveCore very robust to memory design decisions. We demonstrate that both on-chip buffer capacity and available off-chip bandwidth have far smaller impact than using a conventional training approach. For example, even with a low-cost LPDDR4 DRAM system (the same DRAM used for mobile phones), WaveCore can outperform a high-end GV100 GPU.

REFERENCES

Alwani, M., Chen, H., Ferdman, M., and Milder, P. Fused-layer cnn accelerators. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12. IEEE, 2016.

Chen, K., Li, S., Muralimanohar, N., Ahn, J. H., Brockman, J. B., and Jouppi, N. P. Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 33–38. EDA Consortium, 2012.

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices*, 49(4):269–284, 2014a.

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622. IEEE Computer Society, 2014b.

Chen, Y.-H., Krishna, T., Emer, J. S., and Sze, V. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Das, D., Avancha, S., Mudigere, D., Vaidynathan, K., Sridharan, S., Kalamkar, D., Kaul, B., and Dubey, P. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

Dean, J. Machine learning for systems and systems for machine learning. 2017.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y., and Temam, O. Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pp. 92–104. ACM, 2015.

Fuketa, H., Hirairi, K., Yasufuku, T., Takamiya, M., Nomura, M., Shinohara, H., and Sakurai, T. Minimizing energy of integer unit by higher voltage flip-flop: Vddmin-aware dual supply voltage technique. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(6): 1175–1179, 2013.

Gao, M., Pu, J., Yang, X., Horowitz, M., and Kozyrakis, C. Tetris: Scalable and efficient neural network acceleration with 3d memory. *ACM SIGOPS Operating Systems Review*, 51(2):751–764, 2017.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 243–254. IEEE, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hickmann, B., Krioukov, A., Schulte, M., and Erle, M. A parallel ieee p754 decimal floating-point multiplier. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pp. 296–303. IEEE, 2007.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678. ACM, 2014.

High Bandwidth Memory (HBM) DRAM, JESD235A. Joint Electron Device Engineering Council, Jan. 2016.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pp. 1–12. IEEE, 2017.

Kahng, A. B., Li, B., Peh, L.-S., and Samadi, K. Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration. In *Proceedings of the conference on Design, Automation and Test in Europe*, pp. 423–428. European Design and Automation Association, 2009.

Kim, Y., Jung, W., Lee, I., Dong, Q., Henry, M., Sylvester, D., and Blaauw, D. 27.8 a static contention-free single-phase-clocked 24t flip-flop in 45nm for low-power applications. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pp. 466–467. IEEE, 2014.

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

Kung, H.-T. Why systolic architectures? *IEEE computer*, 15(1):37–46, 1982.

Kurzak, J., Tomov, S., and Dongarra, J. Autotuning gemm kernels for the fermi gpu. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 2012.

Li, M., Zhang, T., Chen, Y., and Smola, A. J. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 661–670. ACM, 2014.

Lu, W., Yan, G., Li, J., Gong, S., Han, Y., and Li, X. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 553–564. IEEE, 2017.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaev, O., Venkatesh, G., et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

nvidia. Nvidia tesla v100 gpu architecture. *White paper*, 2017.

Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S. W., and Dally, W. J. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 27–40. ACM, 2017.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pp. 1139–1147, 2013.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, pp. 12, 2017.

Vogelsang, T. Understanding the energy consumption of dynamic random access memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 363–374. IEEE Computer Society, 2010.

Wu, Y. and He, K. Group normalization. *arXiv preprint arXiv:1803.08494*, 2018.