Tangram: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators

Mingyu Gao
mgao12@stanford.edu
Stanford University
Tsinghua University

Xuan Yang
xuany@stanford.edu
Stanford University

Jing Pu
jingpu@stanford.edu
Stanford University
Google, Inc.

Mark Horowitz
horowitz@stanford.edu
Stanford University

Christos Kozyrakis
kozyraki@stanford.edu
Stanford University
Google, Inc.

Abstract

The use of increasingly larger and more complex neural networks (NNs) makes it critical to scale the capabilities and efficiency of NN accelerators. Tiled architectures provide an intuitive scaling solution that supports both coarse-grained parallelism in NNs: *intra-layer parallelism*, where all tiles process a single layer, and *inter-layer pipelining*, where multiple layers execute across tiles in a pipelined manner.

This work proposes dataflow optimizations to address the shortcomings of existing parallel dataflow techniques for tiled NN accelerators. For intra-layer parallelism, we develop *buffer sharing dataflow* that turns the distributed buffers into an idealized shared buffer, eliminating excessive data duplication and the memory access overheads. For inter-layer pipelining, we develop *alternate layer loop ordering* that forwards the intermediate data in a more fine-grained and timely manner, reducing the buffer requirements and pipeline delays. We also make inter-layer pipelining applicable to NNs with complex DAG structures. These optimizations improve the performance of tiled NN accelerators by 2× and reduce their energy consumption by 45% across a wide range of NNs. The effectiveness of our optimizations also increases with the NN size and complexity.

CCS Concepts • **Computer systems organization → Neural networks**; **Data flow architectures**.

Keywords neural networks, parallelism, dataflow

ASPLOS '19, April 13–17, 2019, Providence, RI, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6240-5/19/04...$15.00
https://doi.org/10.1145/3297858.3304014

ACM Reference Format:
Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. Tangram: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators . In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304014

1 Introduction

Neural networks (NNs) are currently the most effective solution for many challenging classification, recognition, and prediction problems [24]. Hence, there is significant interest in finding *scalable and energy efficient* ways to run NNs on devices ranging from datacenter servers to mobile clients.

Recent research has shown that domain-specific NN accelerators can achieve more than two orders of magnitude improvements over CPUs and GPUs in terms of performance and energy efficiency [1, 4, 5, 7, 9, 11, 12, 15, 28, 33, 36]. NN accelerators typically use spatial architectures with 1D or 2D arrays of processing elements (PEs) and on-chip SRAM buffers to facilitate data reuse. The software that orchestrates the dataflow between the on-chip and off-chip memories and the PEs is also critical in achieving high performance and energy efficiency [6, 11, 14, 28].

The need for higher accuracy on increasingly complex problems leads to larger NNs with higher compute and memory requirements. For example, recent NNs use up to a few hundreds of layers, with each layer sized at several megabytes [17, 39, 42]. Hence, it is important to scale up NN accelerators to efficiently support larger NNs. An intuitive approach for scalable acceleration is to use *tiled architectures*, where each tile includes a small 2D PE array and a local SRAM buffer [5, 14, 22, 44]. A network-on-chip interconnects the tiles. To get scalable performance on tiled accelerators, we must optimize the *coarse-grained parallelism* across multiple tiles, in addition to the *fine-grained parallelism* within each engine. Existing dataflow schemes for coarse-grained parallelism suffer from significant inefficiencies. Parallelizing a single NN layer (*intra-layer parallelism*) leads to significant data duplication [14, 22], and pipelining the processing of multiple layers (*inter-layer pipelining*) results in substantial challenges in resource utilization and on-chip buffer requirements [25, 40].

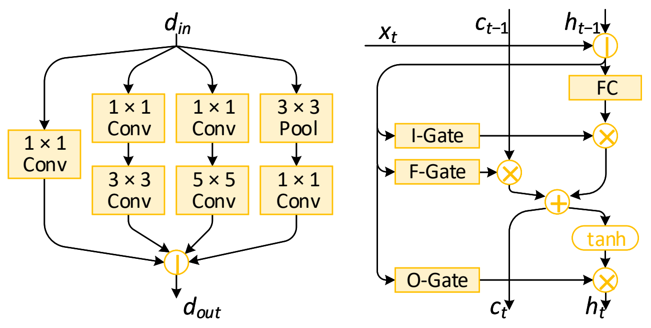
To overcome these challenges, we present TANGRAM, a scalable tiled accelerator with novel dataflow optimizations for coarse-grained parallelization of NN workloads. TANGRAM uses the same hardware resources as prior tiled NN architectures. Its primary contribution is the optimized dataflow. For intra-layer parallelism, we develop *buffer sharing dataflow (BSD)* that eliminates the inefficiencies resulted from data duplication in on-chip buffers. It effectively turns the distributed SRAM buffers into an idealized shared buffer that always has the necessary data close to the processing tile. For inter-layer pipelining, TANGRAM introduces *alternate layer loop ordering (ALLO) dataflow* to forward intermediate fmap data in a more fine-grained and timely manner, which reduces the on-chip buffering requirements and the pipeline filling/draining delays. TANGRAM also includes pipelining optimizations for the complex DAG structures in advanced CNNs and LSTMs, which can minimize the number of complex data dependencies served through the off-chip memory. This extends the applicability of inter-layer pipelining beyond the simple linear NNs targeted in previous work.

We evaluate TANGRAM using large-scale, state-of-the-art, CNN, MLP, and LSTM models. We show that by using optimized parallel dataflow, TANGRAM improves upon an already optimized baseline with the same tiled hardware, with 2.0× higher performance and 45% less energy. These benefits allow TANGRAM to sustain 6107.4 GOPS performance and 439.8 GOPS/W energy efficiency. These numbers represent an efficiency improvement equivalent to two technology node generations. We also perform a detailed analysis of the intra-layer and inter-layer dataflow optimizations to understand their individual contributions. The effectiveness of these optimizations increases for larger NNs (intra-layer) and NNs with complex DAGs (inter-layer). Hence, we believe that TANGRAM represents an effective way to accelerate advanced NNs in the future.

2 Background

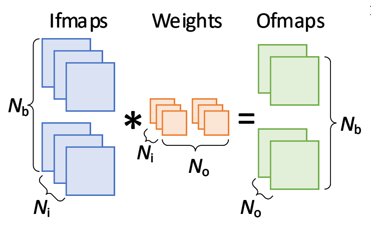
2.1 Neural Network Algorithms

Deep (DNNs), Convolutional (CNNs), and Recurrent Neural Networks (RNNs) are the most widely used NNs today. The typical NN structure is a directed acyclic graph (DAG) composed of multiple *layers*. While DNNs and many CNNs use a single linear chain, RNNs, such as Long Short-Term Memories (LSTMs), and advanced CNNs exhibit the complex DAG structures shown in Figure 1. During *inference*, data propagate in the forward direction, from the first layer that accepts the input (e.g., image, text), to the last layer that produces the result (e.g., image label, translated text). During *training*, data first propagate forward to generate an inference result to compare against the ground truth, and then the errors propagate backward to update the model weights in each layer.



(a) GoogLeNet inception module. **(b)** LSTM cell.

Figure 1. Complex NN DAG structures.



```
for sample $b \leftarrow 0$ to $N_b$
  do
    for ifmap $i \leftarrow 0$ to $N_i$
      do
        for ofmap $o \leftarrow 0$ to $N_o$
          do
            O[$b$][$o$] +=
                I[$b$][$i$] $*$ W[$o$][$i$]
```

Figure 2. CONV layer computation. N_i ifmaps are convolved with different filter weights and accumulated to N_o ofmaps. Computations are performed in batch size of N_b.

The most common NN layer types are fully-connected (FC) and convolutional (CONV). The gates in LSTM cells are essentially FC layers. An FC layer generates a 1D vector output by performing matrix-vector multiplication between its weight matrix and the input vector. The output of a CONV layer is organized as multiple 2D feature maps (*fmaps*), as shown in Figure 2. Each output fmap (ofmap) is the sum of 2D convolutions between all input fmaps (ifmaps) and a set of filter weights. To amortize the cost of weight accesses, NN computations are often performed on a *batch* of data samples. Since an FC layer can be viewed as a CONV layer with 1 × 1 fmaps, the computation of both FC and CONV layers can be summarized as:

$$\mathbf{O}[b][o] = \sum_{i=0}^{N_i-1} \mathbf{I}[b][i] * \mathbf{W}[o][i] + \mathbf{B}[o],\ \ 0 \le o < N_o,\ \ 0 \le b < N_b \quad (1)$$

where **I** and **O** are the 4D ifmaps and ofmaps (2D image, number of fmaps, and batch), **W** is the filter weights, and **B** is a 1D bias. "$*$" denotes 2D convolution. N_i, N_o, N_b are the number of ifmaps, ofmaps, and the size of batch, respectively.

NNs also include other types of layers. CONV and FC layers are typically followed by activation (ACT) layers, which apply non-linear functions such as ReLU or sigmoid. Maximum or average pooling (POOL) layers are optionally added

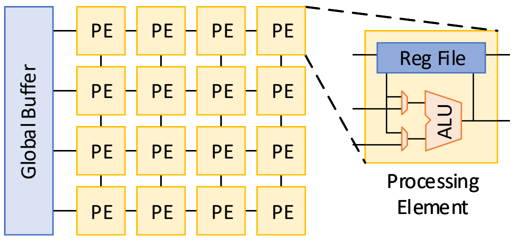


Figure 3. NN engine with a 2D PE array and a shared buffer.

between CONV layers to reduce fmap dimensions. ACT and POOL layers do not have weights, thus data can be easily processed in a streaming fashion.

For training, the forward computation is the same as inference. The most common algorithm for backward propagation is gradient descent. In CONV layers, the errors are convolved with the previously generated ofmaps; in FC layers, the errors are multiplied with the output vectors. They can be formulated as CONV and FC layers with different dimensions [40, 44]. Because the output of each layer is needed for backward propagation, they must be written to off-chip memory during forward computation and fetched back later.

2.2 NN Accelerators and Dataflow Scheduling

Several accelerator designs have been developed to address the high compute and memory requirements of NNs (see Section 7). We use the state-of-the-art Eyeriss accelerator as our baseline *NN engine* [7]. As shown in Figure 3, the Eyeriss NN engine includes a number of processing elements (PEs) organized in a 2D array. Each PE contains a simple ALU for multiply-accumulate (MAC) operations and a small register file of 64 to 512 bytes. A larger SRAM buffer is shared by all PEs. Other NN accelerators use a similar architecture [1, 4, 5, 11, 12, 28].

The efficiency of NN engines depends on how the nested loops in NN computations shown in Figure 2 are scheduled [6, 14, 46]. Since the total data size for NNs is too large to fit entirely on-chip, the on-chip and off-chip dataflows are crucial for performance. *Loop transformations*, such as blocking and reordering, can maximize data reuse across loop iterations in the on-chip buffer and minimize accesses to the off-chip memory [46]. *Array mapping* techniques optimize the spatial mapping of the 2D convolution for each pair of ifmap $\mathbf{I}[b][i]$ and ofmap $\mathbf{O}[b][o]$ on the PE array, in order to maximize parallelism and capture the locality in the PE registers [6].

2.3 Parallelizing NN Acceleration

Since large NNs with more layers and more complex DAG structures provide higher accuracy on more challenging tasks [17, 18, 42], there is strong interest in scaling NN accelerators. Simply increasing the number of PEs in the monolithic engine in Figure 3 is not efficient. First, small layers

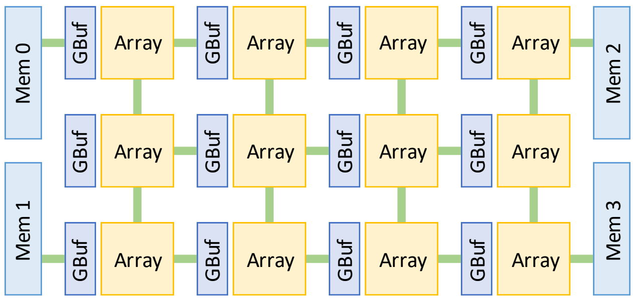


Figure 4. 2D tiled architecture with multiple NN engines connecting to off-chip memory channels on the sides.

Table 1. Data sharing characteristics of different NN coarse-grained parallelization schemes.

Scheme	Ifmaps	Ofmaps	Weights
Intra-layer			
Batch	Partitioned¶	Partitioned¶	*Replicated*
Fmap	With overlaps*	Partitioned*	*Replicated*
Output	*Shared*	Partitioned†	Partitioned†
Input	Partitioned†	*Shared*	Partitioned†
Inter-layer			
Pipeline	*Forwarded*	*Forwarded*	Separate

¶ Partitioned across batch samples.
* Partitioned inside one fmap.
† Partitioned across different fmaps.

cannot fully utilize a large PE array, as computations sharing the same data are typically placed on one PE to exploit register locality. While one can map multiple 2D convolutions to a single PE array [6], these convolutions are independent and can only result in interference. Second, larger PE arrays incur higher latency and energy overheads for the data multicast needed when each time we start processing a new set of fmaps [8] (see Figure 11). Third, the increasing distance between the shared buffer and the PEs also becomes a bottleneck. While banking can help, most PEs would still be quite far from the banks they need to access. Finally, a monolithic array with rigid PE interconnects cannot support the dataflow for inter-layer pipelining.

An efficient approach to get scalable performance is to build a tiled architecture with multiple NN engines [5, 14, 22, 40, 44]. As shown in Figure 4, engines communicate through a network-on-chip (NoC) that also connects them to off-chip memory channels. The tiled architecture allows for *coarse-grained parallelism*, where NN computations are coarsely parallelized onto different engines. This is in addition to the *fine-grained parallelism* of the spatially mapped 2D convolutions on the PE array in each engine. To make an analogy to general-purpose processors, fine-grained parallelism corresponds to SIMD or instruction-level parallelism, while coarse-grained parallelism corresponds to multi-core.

There are two types of coarse-grained parallelism for NN computations. First, multiple engines can process in parallel the computations of a single layer [5, 14, 22]. Table 1 shows different schemes to leverage such *intra-layer parallelism* [14]. Batch parallelization partitions the batch so that each engine processes different data samples (data parallelism). Fmap parallelization tiles the i/ofmaps and uses each engine to process a sub-tile region of all fmaps. Output parallelization parallelizes the ofmap loop in Figure 2 and uses each engine to process a subset of the ofmaps. Similarly, input parallelization parallelizes the ifmap loop.

Alternatively, we can use multiple engines to process multiple layers in a pipelined manner by spatially mapping the NN DAG structures [2, 25, 36, 38, 40, 44]. Such *inter-layer pipelining* is effective in increasing the hardware utilization when layers are small and hardware resources are abundant. This is the case for many recent NNs that use large numbers of layers but each individual layer is rather small [17, 18, 45]. With inter-layer pipelining, the intermediate fmaps can be forwarded between layers through the on-chip NoC, reducing the energy cost for off-chip memory accesses.

2.4 Baseline Architecture and Its Inefficiencies

We focus on optimizing intra-layer parallelism and inter-layer pipelining on tiled NN architectures. Existing, state-of-the-art techniques have significant inefficiencies.

Baseline hardware: We start with an optimized tiled NN accelerator (Figure 4) similar to recent proposals [14, 44]. We consider 16×16 tiles, where each tile is an Eyeriss-like NN engine that includes an 8×8 PE array and a 32 kB private SRAM buffer [7] (Figure 3). We leverage the state-of-the-art row stationary dataflow for PE array mapping [6], and the loop transformation techniques by Yang, et al. to optimally manage SRAM buffers [46]. When switching layers, we elide the off-chip accesses (no writeback, no refetch) if the intermediate fmaps can fit entirely in on-chip buffers.

Baseline intra-layer dataflow: The baseline system supports intra-layer parallelism using hybrid output and fmap parallelization as proposed in TETRIS [14]. All engines fetch data in parallel from multiple off-chip memory channels. If input data is shared between engines, they are fetched from memory once and broadcast. Similarly, output data are fully accumulated across engines and only the final results are written back to memory, following the dataflow in ScaleDeep [44].

This approach uses the buffer within each engine as a private cache that holds a full copy of any shared data. Such data duplication can waste significant amounts of overall SRAM buffer capacity. For example, duplicating a moderate size of 64 kB data across the 16×16 tiles could result in a waste of 16 MB buffer space. This scenario is likely to happen. Table 1 shows that none of the intra-layer parallelization schemes can fully partition all data types among NN engines. At least one data type (ifmaps, ofmaps, or weights) is shared and thus must be duplicated (denoted in italics). Data duplication reduces the effective capacity of on-chip buffers, which leads to reduced reuse opportunities and more off-chip accesses. For example, duplication may prevent the intermediate fmaps between two layers from fitting in the on-chip SRAM. Also, duplication leads to significantly larger area requirements for the accelerator chip with larger SRAM buffers.

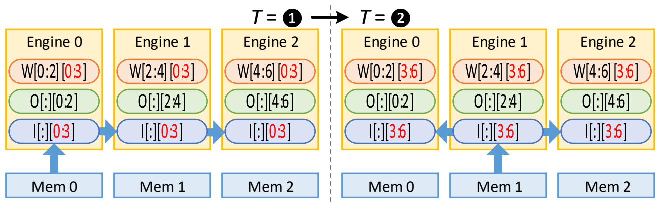
Baseline inter-layer pipelining: Previous proposals for inter-layer pipelining assumed sufficient hardware resources, so that the entire NN (all layers) can be mapped onto a single or multiple chips [25, 40, 44]. This approach does not scale to large NNs with hundreds of layers.

Our baseline system supports inter-layer pipelining by dividing the layers of large NNs into *segments*. At each time, only a single segment of layers is scheduled on the tiled architecture. On-chip resources are allocated to the layers in the segment proportional to their total number of operations [44]. Only the first layer input and the last layer output in the segment require off-chip accesses. The intermediate fmaps are directly forwarded through the on-chip buffers between layers. In addition, when all layers in the NN spatially execute on different engines and their weights fit entirely in the on-chip buffers, we support model weight pinning and avoid accessing weights from the off-chip memory [13].

Still, the baseline approach has several inefficiencies. First, direct forwarding intermediate fmaps requires large on-chip buffers. The entire fmaps must be stored on-chip and double buffered for concurrent read and write accesses from adjacent layers [44]. This translates to tens or hundreds of MBytes of on-chip SRAM (see Table 2), and can only be made worse by the data duplication in intra-layer dataflow. Second, when switching between segments, the hardware resource utilization drops substantially due to the overheads of filling and draining the segment pipelines. Later layers in the segment cannot start processing until the previous layers produce their output data. Finally, all prior work has limited the inter-layer pipelining to linear NN structures, with no support for complex DAGs.

3 Tangram Parallel Dataflows

We propose Tangram, a tiled NN architecture with novel dataflow optimizations for intra-layer parallelism and inter-layer pipelining. Tangram uses the same hardware resources as the baseline, but its novel parallel dataflow overcomes the buffer requirement and resource utilization challenges in previous designs (Sections 3.1 and 3.2). It also extends inter-layer pipelining to support complex DAG structures in advanced CNNs and LSTMs (Section 3.3). Tangram is designed for inference tasks, but its dataflow optimizations can be similarly applied to training (Section 3.4). This section focuses on the dataflow optimizations. We discuss their hardware and software implementations in Section 4.


(a) Baseline dataflow.

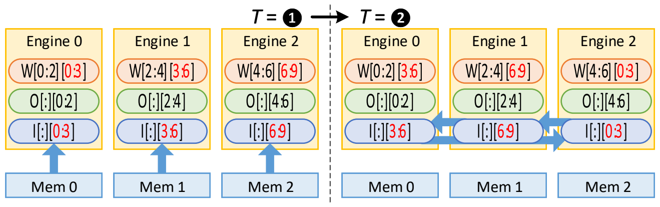

(b) Buffer sharing dataflow.

Figure 5. Intra-layer dataflow with output parallelization sharing ifmaps. **I**$[b][i]$ means the ith ifmap of the bth sample. **W**$[o][i]$ means the weights for the ith ifmap and oth ofmap.

3.1 Intra-Layer Parallelism with Buffer Sharing

We start by improving the dataflow across multiple NN engines processing a single layer. Figure 5(a) reviews the dataflow in the baseline (Section 2.4). In this example, we consider output parallelization, where each engine processes a different ofmap subset using the corresponding weights. The same set of ifmaps are shared and duplicated in the buffers of all engines (e.g., first at time ❶ **I**[:][0 : 3] from memory 0, then at time ❷ **I**[:][3 : 6] from memory 1). When parallelizing over p_o engines, there are p_o copies of the same ifmaps in the SRAM. Similarly, with fmap or batch parallelization, the shared weights are duplicated; with input parallelization, the shared ofmaps are duplicated. This duplication wastes expensive on-chip buffers and fails to benefit from data reuse patterns both within one layer and across adjacent layers.

Buffer sharing dataflow (BSD): BSD is a parallel dataflow optimization that eliminates shared data duplication across engines. With BSD, each engine contributes its buffer capacity to store a subset of the shared data and continuously exchanges data with other engines, until all shared data have passed through and been processed locally by all engines.

Figure 5(b) illustrates BSD. We first *skew* the computation order across the engines to make each engine fetch and process a different subset of the shared data from the nearby memory channel. At time ❶, engine 0 starts with **I**[:][0 : 3], engine 1 starts with **I**[:][3 : 6], etc. Since the ofmap accumulation is commutative, skewing the order does not affect the accuracy. When all engines finish processing their current subsets of data, we *rotate* the shared data in order to allow each engine to process a different subset. At time ❷, after the first rotation, engine 0 processes **I**[:][3 : 6] sourced from engine 1, engine 1 processes **I**[:][6 : 9] sourced from engine

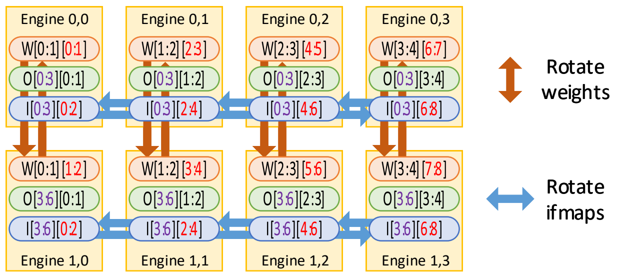


Figure 6. Buffer sharing dataflow with 2D data rotation for hybrid output and fmap/batch parallelization. First rotate weights vertically; then rotate ifmaps horizontally.

2, and engine 2 processes **I**[:][0 : 3] sourced from engine 0. Further rotation steps ensure that all ifmaps pass through all engines and are fully processed. We then fetch the next ofmap subset and rotate the same ifmaps for another round (not shown in Figure 5(b)), until the ifmaps are fully used to update all ofmaps.

BSD is also applicable to weight sharing with fmap or batch parallelization, as well as hybrid parallelization as shown in Figure 6. With hybrid ifmap and weight sharing, data are logically distributed in 2D, and rotation also happens in 2D. We first rotate the weights vertically to complete processing the fmaps currently in the local buffers (e.g., **I**[0 : 3][0 : 2] and **O**[0 : 3][0 : 1] in engine (0, 0)), and then rotate the ifmaps horizontally to obtain new ranges of ifmaps (e.g., engine (0, 0) gets **I**[0 : 3][2 : 4] from engine (0, 1)).

Loop transformation model for BSD: We represent BSD using the loop transformation model in [46]. As shown in Figure 7, originally the on-chip buffer of each engine can store N_i/t_i ifmaps and N_o/t_o ofmaps, with $1/t_b$ of the batch N_b, corresponding to the top-level loop blocking factors. Each time the engine fetches a subset of ifmaps, ofmaps, and weights for on-chip processing according to the loop indices. With BSD, the p_o engines with output parallelization now buffer different ifmaps, p_o times larger than before, reducing the ifmap loop factor by p_o. To rotate the ifmaps between engines, an additional blocking loop level i_0'' is introduced, and skewed by the engine index x. Because the rotation loop is inside the ofmap loop, the $N_i/(t_i/p_o)$ ifmaps are rotated $r = t_o$ rounds on-chip and fully reused by all N_o ofmaps.

In general, assume that N data are shared by p engines (e.g., N_i ifmaps shared by p_o engines), and that each engine buffer stores $1/t$ of the shared data. If the total number of rotation rounds is r decided by the outer loop factors, the subset index of the shared data in the xth engine at time step T will be

$$i_0 = \lfloor \frac{T}{rp} \rfloor \times p + (x + T \bmod p) \bmod p, \ 0 \le T < \lceil \frac{t}{p} \rceil \times r \times p \quad (2)$$

In the case of hybrid parallelization, the index of each shared data can be calculated independently.

```
for $b_0 \leftarrow 0$ to $t_b$ do
  for $i_0 \leftarrow 0$ to $t_i$ do
    // fetch ifmap subset [$b_0$][$i_0$].
    for $o_0 \leftarrow 0$ to $t_o$ do
      // fetch ofmap subset [$b_0$][$o_0$], weight subset
         [$o_0$][$i_0$].
      // on-chip processing.
```

(a) Loop blocking without BSD.

```
for $b_0 \leftarrow 0$ to $t_b$ do
  for $i'_0 \leftarrow 0$ to $t_i/p_o$ do
    // fetch ifmap subset [$b_0$][$i'_0$] into $p_o$ engines.
    for $o_0 \leftarrow 0$ to $t_o$ do
      // fetch ofmap subset [$b_0$][$o_0$], weight subset
         [$o_0$][$i_0$].
      for $i''_0 \leftarrow 0$ to $p_o$ do
        $i_0 = i'_0 \times p_o + (x + i''_0) \bmod p_o$
        // rotate to get ifmap subset [$b_0$][$i_0$].
        // on-chip processing.
```

(b) Loop blocking with BSD.

Figure 7. Loop transformation model for BSD, for output parallelization that shares ifmaps. Ifmap data rotation is orchestrated by the additional loop level i''_0.

Using the above model, the compiler can statically analyze the dataflow and fully manage the hardware at runtime. We present further implementation details in Section 4.

BSD benefits: BSD optimizes the use of SRAM buffers *across* all engines. With ifmap sharing as an example, we can rotate the ifmaps multiple rounds, each for a different set of ofmaps. This allows for the reuse of a larger set of ifmaps (those across all engines, $\mathbf{I}[:][0:9]$, rather than those in a single engine, $\mathbf{I}[:][0:3]$, in Figure 5(b)) across all the ofmaps without the need for off-chip accesses. With the loop blocking shown in Figure 7, the number of ofmap off-chip fetches is decided by the outer level ifmap loop factor. Previously the ofmaps are fetched t_i times, each being updated using N_i/t_i ifmaps. With BSD, the ofmaps are updated with $p_o\times$ more ifmaps each time, and thus only fetched t_i/p_o times from off-chip.

BSD also improves data reuse when switching between adjacent layers. Without BSD, each engine needs a private copy of the input data from the previous layer. If these ifmaps do not fit in the buffer of a *single* engine, we have to spill them using off-chip memory. By eliminating data duplication with BSD, as long as the ifmaps can fit on-chip using *all* SRAM buffers, we can directly reuse the buffered intermediate fmaps from the previous layer, and elide off-chip accesses when switching layers.

In fact, BSD is equivalent to the ideal case where a single, large buffer with the aggregate capacity of all engine buffers stores all data with no duplication. So it achieves the maximum on-chip data reuse. Moreover, by combining computation skew and data rotation, we ensure that the data currently being processed always reside in the local buffer and do not need to be accessed remotely. Hence the buffers operate like an optimal NUCA cache [3, 16]. Data rotation happens between neighbor engines, which minimizes the number of NoC hops.

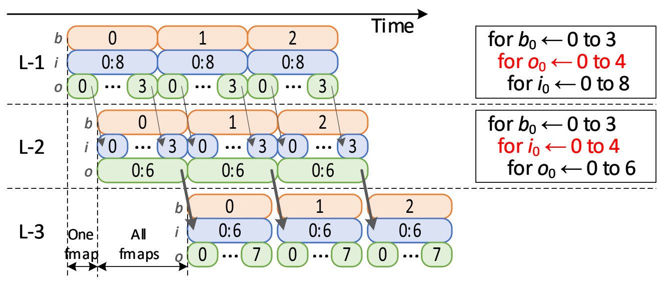

Time
L-1
L-2
L-3
b
i
o
One fmap
All fmaps
for $b_0 \leftarrow 0$ to 3
for $o_0 \leftarrow 0$ to 4
for $i_0 \leftarrow 0$ to 8
for $b_0 \leftarrow 0$ to 3
for $i_0 \leftarrow 0$ to 4
for $o_0 \leftarrow 0$ to 6

Figure 8. Timing diagram of inter-layer pipelining with alternate layer loop ordering (ALLO). Orange, blue, and green boxes denote top level batch, ifmap, ofmap loops, respectively. Arrows indicate data dependencies and inter-layer data forwarding. The top level loop blocking of each layer is shown on the right. Matched fmap access patterns are denoted in red.

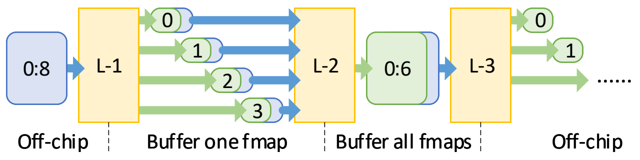


Figure 9. Intermediate fmap buffering of inter-layer pipelining with alternate layer loop ordering (ALLO). Blue and green boxes denote ifmaps and ofmaps of each layer, respectively.

3.2 Inter-Layer Pipelining with ALLO

As discussed in Section 2, while inter-layer pipelining can improve overall performance and resource utilization in the presence of small layers, it requires significant on-chip buffer capacity to hold the intermediate fmap data between layers. Moreover, data dependencies between layers result in long pipeline filling/draining delays that degrade performance.

A simple approach to alleviate these issues is to break up the input data batch of each pipeline stage into multiple subsets. Instead of waiting for its entire input to be available, each layer can start processing as soon as a subset of its input samples are ready. This approach requires that the outermost loop of each layer is blocked by the same factor t_b. It reduces on-chip buffer requirements and pipeline filling/draining delays to $1/t_b$. However, t_b is constrained by the total batch size N_b, which is small for inference tasks. It also sacrifices the weight reuse, as the weights must be fetched t_b times (once per each data subset). Therefore, t_b can only be moderate, around 4 to 8, leading to limited savings.

Alternate layer loop ordering (ALLO) dataflow: We propose a novel pipelining dataflow, called *Alternate layer loop ordering* (ALLO), that further reduces the pipeline delays and buffer requirements on top of breaking up the data batch. ALLO modifies the intermediate fmap data access patterns, in order to allow the next layer to start processing as soon as a subset of the fmaps *within a single data sample* are ready. For example, in Figure 8, L-1 computes each ofmap sequentially. If the next layer (L-2) also sequentially accepts these data as its ifmaps, it can start processing after waiting for a single fmap rather than all fmaps. Since each ofmap generated by L-1 is immediately used as an ifmap by L-2, we only need to store a single fmap that is currently being processed, rather than all fmaps, as Figure 9 shows.

ALLO dataflow makes two adjacent layers in the pipeline segment access their shared intermediate fmaps with the same sequential pattern, in order to forward and buffer the fmaps in a finer granularity. However, because CONV and FC layers cannot have sequential accesses to both the ifmaps and ofmaps, it is only possible to apply ALLO to alternate pairs of adjacent layers in a pipeline segment. For example, in Figures 8 and 9, while L-1 and L-2 can be optimized with ALLO, the ofmaps of L-2 must be updated multiple times with the sequentially accessed ifmaps, therefore they must be fully buffered, and cannot benefit from ALLO.

Loop transformation model for ALLO: Similar to BSD, ALLO can also be realized using loop transformation techniques. We notice that, sequentially accessing the i/ofmaps corresponds to putting the i/ofmap loop at the outer level, right below the top batch loop required by breaking up batches. Therefore, ALLO requires the adjacent layers in the pipeline segment to use alternate orders for the ifmap loop and the ofmap loop. To enforce exactly the same subsets, the loop blocking factors should also match. As shown in Figure 8, L-1 and L-2 use alternate loop orders, having the ofmap loop and the ifmap loop at the outer level, respectively (denoted in red). They also use the same blocking factor 4. So every time L-1 will produce one fourth of the ofmaps and immediately forward them to L-2 to be consumed.

ALLO benefits: If two adjacent layers have matched outer i/ofmap loops with blocking factor t, ALLO reduces the pipeline filling/draining delays and the on-chip buffer capacity for intermediate fmaps both by a factor of t. These benefits are on top of breaking up the pipelined data batch. Since CONV and FC layers typically have hundreds of fmaps ($N_\mathrm{i}, N_\mathrm{o} > N_\mathrm{b}$), the savings from ALLO ($t_\mathrm{i}, t_\mathrm{o}$) can be substantially higher than pipelining the batch (t_b).

Nevertheless, ALLO can only be applied to half of the pairs of adjacent layers in a pipeline segment. When there are ℓ layers in the segment and $\ell - 1$ intermediate fmap data, ALLO still requires to fully delay and buffer $\lfloor \frac{\ell-1}{2} \rfloor$ of intermediate data. Segments with two layers are a special case; ALLO can optimize the intermediate fmap dataflow and require no fully delay or buffering.

Combining ALLO and BSD: ALLO is compatible with the BSD optimization for intra-layer parallelism. ALLO orchestrates the dataflow *between* layers, while BSD is applied *within* a layer. Moreover, ALLO requires specific orders and blocking factors *within* the top-level loops for off-chip access (Figure 8), while BSD adds an additional loop level *below* the top level (Figure 7) for on-chip data rotation.

Combining ALLO and BSD enables higher savings in on-chip buffer capacity. In fact, BSD helps most with the half of layers in the pipeline segment that cannot use ALLO. For these layers, the intermediate fmap data must be fully buffered on-chip. BSD ensures no data duplication within these layers, so the required buffer capacity is minimized.

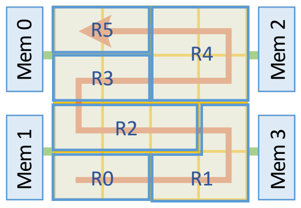
3.3 Inter-Layer Pipelining for Complex NN DAGs

Recent NNs, such as ResNet [17] and various LSTMs [41, 45], feature complex DAG structures that go beyond the single linear chain of layers in early NNs. To support inter-layer pipelining for such complex NN DAG structures, we first improve the allocation strategy, and then provide practical and general heuristics to optimize the spatial mapping of NN layers on the tiled accelerator.

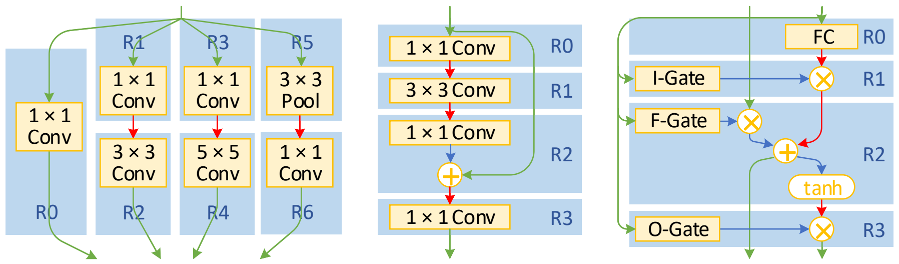
2D region allocation: Prior designs used static 1D allocation strategies to divide on-chip engines into regions that process different layers [44]. Each layer gets one or more columns in the 2D array of NN engines, and fmap data flow in the horizontal direction through all layers. This allocation strategy is not sufficient for NN DAG structures with complex data forwarding patterns. Instead, we propose a 2D zig-zag allocation strategy shown in Figure 10(a). Regions are folded into the next row when they cannot fit in the remaining space in the current row (e.g., R1 and R4).

This 2D allocation strategy has two major advantages. First, it is more fine-grained than 1D allocation and allows to more accurately size the regions to match the computation needs of the layers. Hence, the resources are better utilized and the latencies of pipeline stages are equalized. Second, when the fmap data are forwarded to non-adjacent regions (e.g., R0 to R3), 2D allocation results in shorter distances across the NoC.

Spatial layer mapping heuristics: In complex NN DAG structures, a layer can have multiple predecessor and/or successor layers. Correspondingly, a region may need to receive input data from both on-chip and off-chip sources. Output data may also need to be forwarded to multiple destinations, and possibly also stored back to memory. Hence, it is no longer trivial to determine which subset of layers should be mapped concurrently for pipeline execution (segment selection), and how to map layers within a segment to available



(a) 2D region allocation.



(b) Spatial mapping of complex DAGs. Left: GoogLeNet inception; middle: ResNet module; right: LSTM. Green, red, and blue arrows indicate data dependencies from/to off-chip, neighbor, and local regions.

Figure 10. Inter-layer pipelining support for complex NN DAGs.

regions (region mapping). We propose the following practical heuristics to prune the design space that the compiler has to consider when optimizing layer pipelining for a wide set of NNs (see Section 4.2).

Segment selection: The layers in an NN are considered in the DAG topological order to form pipeline segments. A layer may be added into a segment only if it shares some data dependencies with other layers in the segment. The data dependencies can be either receiving the output data from a layer in the segment as in the simple linear pipelining case, or sharing the input data fetched from off-chip memory with another layer in the segment. The later case is specific to complex DAGs with forks. For instance, the GoogLeNet inception module shown in Figure 10(b) has four layers sharing the off-chip input data (R0, R1, R3, and R5). Spatially mapping them on-chip in a single segment allows us to fetch the shared data only once.

The output data of each layer should be either used exclusively by the other layers in the same segment, or directly stored back to the off-chip memory. There is little benefit to pack half of the consuming layers in the same segment as the producing layer, because the intermediate data still need to be stored back to memory for the other half. It is better to gather all the consuming layers into a new segment, if they cannot fit in the current one. This is the case for GoogLeNet in Figure 10(b). For LSTM cells, we relax this constraint and allow at most one consuming layer to be in a different segment (see Figure 10(b) LSTM cell, R2). This relaxation also helps with training as Section 3.4 will discuss.

Region mapping: We group ACT, POOL, and element-wise layers into the regions of their previous CONV or FC layers. These layers do not need to access any weights and typically have small computation loads. Hence they increase only slightly the compute and buffer needs for each region.

NN layers can be mapped to a single region if they form a linear dependency chain. The engine buffers in the region sequentially store the output fmap data of each layer in the chain, and make them directly available only to the single successor layer. In Figure 10(b), the R2 region of the LSTM cell has a linear chain of four layers (see blue arrows), where the three element-wise layers are merged with the FC F-gate.

The layers that use the same region sequentially can have only one dependency from a neighbor region. This input dependency determines the timing of this region relative to the other regions. In classical linear pipelining, each layer has a single dependency on its predecessor layer in the previous region. With complex DAGs, regions can form a linear chain or a tree of dependencies. We do not allow more than one neighbor dependency per region to avoid conflicting timing requirements that lead to pipeline stalls.

3.4 Dataflow Optimizations for NN Training

While Tangram is primarily designed for NN inference, the intra-layer and inter-layer dataflow optimizations can be applied similarly to NN training. The error backward propagation of CONV and FC layers can be formulated as new CONV or FC layers with different dimensions [40, 44]. The BSD optimization in Section 3.1 can be applied to better parallelize these backward layers. For inter-layer pipelining, training extends the NN DAG structure with more layers for backward propagation. Our improved inter-layer pipelining scheme in Section 3.3 can handle such complex DAG structures. ALLO from Section 3.2 can also be used for both forward and backward pipeline segments. The new backward layers have input data dependencies on the output fmaps of their corresponding forward layers. With the relaxed rule of allowing one consuming layer in a different segment, the forward portion of the NN DAG can still be well pipelined.

4 Tangram Implementation

The intra-layer and inter-layer dataflow optimizations in Tangram require only small hardware changes in tiled NN architectures. Their implementations are mostly software, including (a) a search tool that identifies the optimized parallelization schemes for each NN, and (b) a compiler that produces the code for the selected schemes.

4.1 Hardware Support

Existing tiled NN architectures support limited data forwarding within one layer and across layers [44]. Similar to the prior design, all parallel dataflows in Tangram are statically scheduled. At runtime, each NN engine in the tiled accelerator simply executes the instructions generated by the compiler to access data from SRAM buffers and perform NN computations. The lack of dynamic scheduling greatly simplifies the hardware design.

To implement the intra-layer buffer sharing dataflow, we enhance the engine buffer controllers so that they can issue accesses to both the off-chip memory and the other on-chip engine buffers. Additional control logic orchestrates data rotation and computation skew according to the loop transformation model in Figure 7 and Equation (2). We also need to synchronize the data rotation among the engines to avoid stalls or data overwrites. We leverage the MEMTRACK primitive from ScaleDeep [44], which relies on the hardware buffer controllers to track whether the data has received enough updates before it can be read, and enough read accesses before it can be overwritten. Deadlock is eliminated by transferring the data in units of buffer lines and reserving a few free lines in each buffer. Load imbalance is not an issue as long as the hybrid parallelization partitions the data approximately uniformly [14].

The data forwarding needed for inter-layer pipelining is already available in tiled NN architectures [40, 44]. In Tangram, ALLO forwards the intermediate data in a more fine-grained manner as soon as a subset of fmaps are ready. We use the same MEMTRACK primitive to synchronize the data at the necessary granularity. For complex NN DAGs, on-chip fmaps can be forwarded to multiple destination regions, and one engine may require data from multiple sources.

4.2 Dataflow Design Space Exploration

There are a large number of design options for how to map a multi-layer NN on a tiled accelerator. Recent work has already established the need for hybrid parallelization for intra-layer processing [14]. With inter-layer pipelining, we can either choose a deeper pipeline (longer segments) with fewer engines per layer, or use a shallower pipeline (shorter segments) and give each layer more engines and buffer capacity. For complex NN DAGs, there are additional tradeoffs for segment selection and mapping (see Section 3.3). In fact, the choice of pipeline depth reveals a tradeoff in performance and energy. Deeper pipelines avoid more intermediate off-chip accesses, but reduce per-layer resources and likely make per-layer dataflow suboptimal. The pipeline filling/draining delays also increase with the number of layers per segment, leading to resource underutilization for deeper pipeline.

To manage these tradeoffs, we developed a search tool that explores different intra-layer and inter-layer dataflow schemes for a tiled accelerator. The tool takes the NN topology and the hardware specification as input. It supports common NN types including CNNs, DNNs, and LSTMs. It relies on well-known optimizations for the dataflow within each NN engine [6, 46]. The tool generates a large number of configurations with different inter-layer pipelining, intra-layer parallelism, off-chip loop blocking, and on-chip dataflow. It uses the heuristics discussed in Section 3.3 to trim the design space. Our cost model is similar to [6, 14, 46]. For each individual layer, on-chip and off-chip dataflows are exhaustively searched; across layers, we use a combination of dynamic programming and beam search in the layer topological order. We leverage this tool to compare parallel dataflow schemes for tiled architectures in Section 6. The tool is available at https://github.com/stanford-mast/nn_dataflow.

4.3 Code Generation

The Tangram compiler generates the code for the parallel dataflow selected by the search tool. In addition to the NN topology, the input to the compiler includes the pipeline segment partitioning, the region allocation for each segment, the hybrid intra-layer parallelization scheme, the BSD and ALLO information given as loop transformation models, and the array mapping and loop blocking for each single engine.

Our compiler focuses on the data exchanges between on-chip buffers and the data transfers to/from off-chip memories. The parallel dataflow optimizations in Tangram do not change how the NN engine itself works once data are available in each local buffer. Hence, our compiler can be easily retargeted to tiled accelerators using different engines [21, 27, 44]. Based on the dataflow schemes, the compiler combines the loop transformation models at different levels (ALLO, BSD, and single-engine loop blocking) to get the complete nested loop structure [46]. Then, it generates the data access instructions according to the specific order dictated by the loop structure. The compiler also inserts necessary data synchronization primitives at certain points in the nested loops. The instructions are offloaded to the engine buffer controller, which notifies the PE array to start the computation when all data have been fetched [22].

5 Methodology

Workloads: We evaluate Tangram using four state-of-the-art CNNs from ImageNet ILSVRC, as well as two DNNs (multi-layer perceptrons, MLPs) and two LSTMs in medium (-M) and large (-L) scales, summarized in Table 2. The variety of characteristics in these NNs allows us to explore various tradeoffs in parallel dataflow optimizations. All CNNs have several hundreds of MBytes memory footprints. VGGNet has significantly larger layers than the others. GoogLeNet and ResNet have large numbers of layers. MLPs and LSTMs only contain FC layers and their sizes are dominated by the model weights, with very small fmaps. AlexNet, GoogLeNet, ResNet,

Table 2. Representative NNs for evaluation. Fmap and weight sizes are shown for the largest layer and the entire NN (largest/total) with 16-bit fixed-point data and batch 64.

	CONVs	FCs	Fmap size	Weight size
AlexNet [23]	10	3	17.7/ 95.4 MB	72/116 MB
VGGNet [39]	13	3	392.0/1841.7 MB	196/264 MB
GoogLeNet [42]	57	1	98.0/ 453.4 MB	2/ 13 MB
ResNet [17]	155	1	98.0/4318.5 MB	5/115 MB
MLP-M [9]	-	4	125/220 kB	1.5/2.7 MB
MLP-L [9]	-	4	188/376 kB	2.9/6.1 MB
LSTM-M [45]	-	4	64/ 576 kB	1/ 4 MB
LSTM-L [41]	-	16	125/4125 kB	4/61 MB

and both LSTMs have complex DAG structures as in Figure 1. We use a default batch size of 64, and also explore batch sizes from 1 to 256. Datacenter NN inference accelerators can use batch sizes as high as 200 [21].

System models: We model the NN engine shown in Figure 3 after Eyeriss [7]. Assuming 28 nm technology, the engine runs at 500 MHz. The PE area and power are scaled from [7, 11, 28], assuming 0.004 mm^2 and 1 pJ for each 16-bit MAC. We use McPAT 1.3 to model the area and power of the register files and the SRAM buffers at different capacities, and to calculate the characteristics of the PE array bus wires at different lengths [26]. The NoC power is estimated to be 0.61 pJ/bit per hop [26, 43]. The default configuration for the NN engine contains an 8×8 PE array, with a 64 B register file per PE and a 32 kB shared SRAM buffer.

We evaluate a tiled hardware architecture with a 100 mm^2 cost-effective chip area budget, with 16×16 engines. This results in a total of 16384 PEs and 8 MB on-chip SRAM. The chip connects to four off-chip memory channels of LPDDR4-3200 chips, with a total of 24 Gb capacity and 25.6 GBps bandwidth. The power consumption is calculated using the Micron model [29] and the parameters from datasheets [30].

We use performance in GOPS (giga ops per second) and energy efficiency in GOPS/W as the main comparison metrics. The performance captures both the impact of PE utilization across the NN engines and the impact of off-chip memory accesses. We also model the impact of data multicast latencies from SRAM buffers to PE registers. We aggressively assume separate buses for each data type and that each bus can multicast eight data elements per cycle [7]. We have validated the performance model against cycle-accurate memory access trace simulations with zsim [35] and DRAMSim2 [34].

6 Evaluation

We start with an overall comparison of Tangram against the baseline systems in Section 6.1, followed by a detailed analysis of the parallel dataflow optimizations in Section 6.2. In Section 6.3 we further investigate other batch sizes and hardware configurations.

6.1 Overall Tangram Comparison

Figure 11 shows the energy and performance comparison over all evaluated NNs on the three systems with the same hardware resources. The monolithic engine (M) organizes all PEs in a single 128×128 array, with a heavily banked global buffer of 8 MB. The baseline architecture (B) and Tangram (T) both tile the resources into 16×16 smaller engines as shown in Figure 4. They support both intra-layer parallelism and inter-layer pipelining. The baseline uses the techniques summarized in Section 2.4, while Tangram uses the optimizations presented in Section 3. All three systems support direct fmap reuse across adjacent layers without DRAM writeback as long as the intermediate fmaps fit on-chip. The two tiled architectures also allow model pinning.

Compared to the monolithic and baseline systems, Tangram improves on average the performance by 7.2× and 2.0×, and saves 41% and 45% system energy, respectively. Tangram sustains 6107.4 GOPS performance with 88 mm^2 at 28 nm, and achieves 439.8 GOPS/W energy efficiency including off-chip memories, or 936.4 GOPS/W for the chip itself. These numbers are on par with ScaleDeep, a recent NN accelerator in 14 nm [44]. It demonstrates that optimizing intra-layer and inter-layer dataflow in software can provide energy efficiency improvements equivalent to two hardware technology node generations.

The monolithic engine spends significant energy on the array buses (up to 20%), and its performance is also limited primarily by the high latency of data multicast on these long buses. On the other hand, since all on-chip SRAM resources are aggregated into a single global buffer, it is quite effective at capturing reuse within and across layers, resulting in low energy consumption for off-chip memories. However, we do not partition the monolithic buffer to store the weights from multiple layers simultaneously on-chip, so model pinning is not supported (e.g., for MLP-M).

The two tiled architectures use multiple but smaller PE arrays, so the overheads of the array bus within each engine are reduced. However, the baseline suffers from the increased pressure for buffer capacity due to significant data duplication, and the delays of filling/draining pipelines. Both limit the effectiveness of parallelization. Therefore, it consumes substantially higher energy on the off-chip memory and the NoC, in particular for the CNNs with large intermediate fmap data. In contrast, Tangram uses the optimized BSD and ALLO dataflows for intra-layer and inter-layer parallelism, and supports pipelining of complex NN DAGs such as GoogLeNet and LSTMs. These optimizations reduce off-chip accesses, resulting in energy and performance benefits.

Notice that for MLPs and LSTMs, the energy efficiency is dominated by the weight access through the memory hierarchy in the monolithic engine. The two tiled architectures

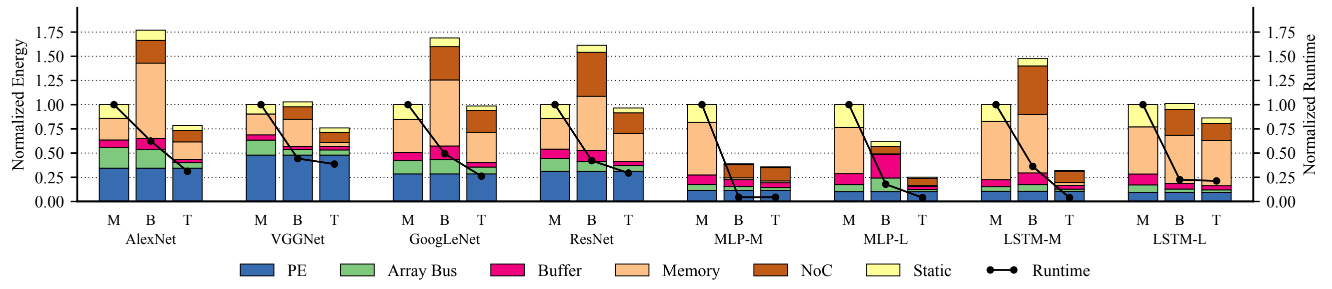


Figure 11. Comparison between monolithic architecture, baseline tiled architecture, and TANGRAM. All three designs use 16384 PEs and 8 MB buffers. In the two tiled architectures resources are organized into 256 tiled engines (16 × 16).

support model pinning when the weights of all layers can fit in the on-chip SRAM, which fully eliminates off-chip weight access and greatly improves efficiency. This is the case for MLP-M and MLP-L in Figure 11. TANGRAM uses dataflow optimizations to reduce the buffer capacity requirements, which enables additional model pinning opportunities such as for LSTM-M. The reduced pressure also allows MLP-L to use more optimized dataflow within each layer, and reduces its energy consumption in the engine buses and buffers. Finally, LSTM-L is too large to use model pinning even with TANGRAM optimizations.

6.2 Parallel Dataflow Analysis

In order to better understand the effectiveness of intra-layer and inter-layer dataflow optimizations proposed in Section 3, Figure 12 compares the energy consumption of TANGRAM against two systems without the two sets of optimizations, respectively. For intra-layer parallelism, all CNNs significantly benefit from the buffer sharing dataflow (BSD) due to their large fmap sizes. Eliminating duplication for the large fmaps greatly saves buffer spaces and improves data reuse within and across layers. BSD also helps with MLP-L and LSTM-M, because the reduced buffer pressure allows more optimized per-layer dataflow to be used with the pipeline schemes, decreasing the buffer access and array multicast energy cost (see Section 4.2). On the other hand, MLP-M and LSTM-L exhibit limited benefits from BSD.

With layer pipelining optimizations, AlexNet, GoogLeNet, ResNet, and the two LSTMs enjoy substantial energy savings. Even with complex DAG structures, these NNs are able to use deeper pipeline segments with more layers in TANGRAM, since ALLO reduces the intermediate data buffering requirements. First, deeper pipelining eliminates more DRAM accesses of the intermediate data. Second, by allocating a smaller region for each layer, the NoC energy for intra-layer traffic is reduced. Third, by using fewer spatial PEs, more operations are co-located to increase PE-level data reuse, which also reduces the bus and buffer energy. In particular, simultaneously pipelining all layers in LSTM-M enables model pinning, greatly saving the energy by 4.6×. Finally, VGGNet and the two MLPs do not benefit much from TANGRAM inter-layer optimizations, as they use only simple linear topologies.

6.3 Hardware and Batch Size Scaling

Figure 13(a) shows the energy of TANGRAM relative to the baseline as we scale from 16 to 1024 engines (tiles). The benefits of TANGRAM increase when more hardware resources are available and are organized into more tiles. In this case, it becomes more difficult for a single layer to utilize all resources, and the inefficiencies of the baseline system mentioned in Section 2.4 become more critical. TANGRAM successfully addresses these issues, and achieves up to 67% energy saving when scaling to 1024 engines with 65536 PEs.

Figure 13(b) shows the impact of batch sizes. The tiled baseline actually performs worse than the monolithic engine at large batches, because the inefficient resource utilization becomes more serious when the data size becomes larger. TANGRAM allows for more efficient data reuse and pipelining with more independent data samples, enabling higher energy improvements with larger batches.

For small batches, TANGRAM can still provide benefits, although the savings become smaller. When using batch size 1, which is common in latency-sensitive inference scenarios [13]. TANGRAM slightly improves the energy efficiency for most NNs such as AlexNet by roughly 10% compared to the tiled baseline. GoogLeNet exhibits more significant energy saving because multiple layers in the pipeline segment share the input, and LSTM-M benefits from model pinning with TANGRAM which eliminates the dominant weight access (not shown in the figure).

7 Related Work

NN accelerators: The importance of NNs has motivated a number of accelerators with 1D inner-product engines [4, 5] or 2D spatial PE arrays [7, 11, 12, 21, 28] with low-precision arithmetics and small control overheads. TANGRAM uses a similar architecture as tiled accelerators [5, 14, 22, 44]. Recent work has also prototyped NN accelerators on FPGAs [2, 25,

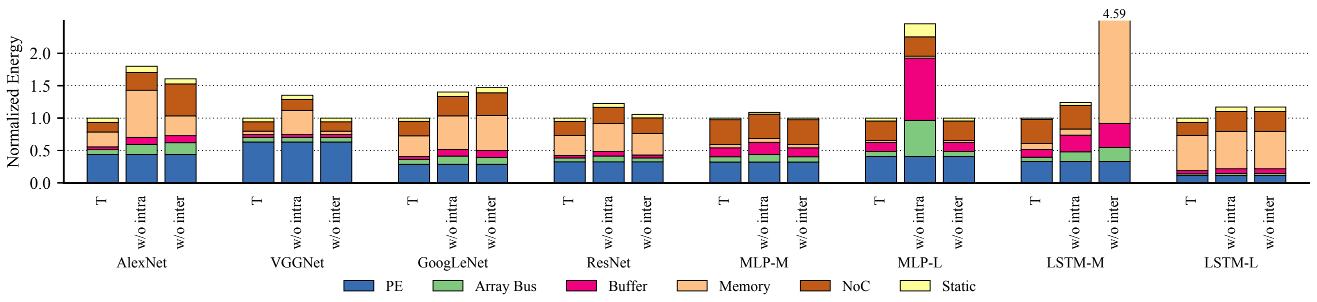
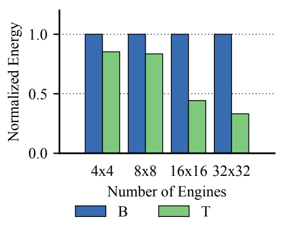
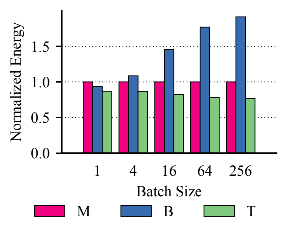


Figure 12. Comparison between TANGRAM and two systems that disable intra-layer and inter-layer optimizations, respectively.



(a) Hardware resources.



(b) Batch sizes.

Figure 13. Effectiveness of TANGRAM optimizations using different numbers of engines and batch sizes for AlexNet.

32, 37, 38, 48], and leveraged ReRAM crossbars for in-situ analog dot-product operations [9, 36, 40].

A notable technique to further improve NN efficiency is to exploit sparsity in NN workloads in order to avoid redundant operations and reduce memory footprints. Recent work either dynamically pruned zero and small values [1, 33], or statically compressed the NN structures into sparse formats [10, 15, 47, 49]. While TANGRAM focuses on dense NNs, its insights should be useful for scaling sparse NN accelerators as well. In fact, sparsity mostly affects the fine-grained parallelism by changing the dataflow inside the engines [31]. The coarse-grained parallelism remains similar to the dense case. We will explore this issue in details in future work.

Intra-layer parallelism: DaDianNao was a tiled architecture with on-chip eDRAM [5]; however the dataflow between tiles was neither thoroughly studied nor optimized. NN accelerators with 3D memory associate one NN engine to each of the 3D channels. Neurocube proposed a simple heuristic for NN partitioning across tiles [22] and TETRIS extended it to hybrid partitioning schemes [14]. They both suffered from the inefficiencies discussed in Section 2.4 and modeled in our baseline system. Our BSD proposal shares similar insights with non-uniform cache access (NUCA) [3, 16]. It leverages application-specific knowledge of NNs to statically schedule computation skew and data rotation in order to optimally migrate data between tiles.

Inter-layer pipelining: ISAAC [36] and PipeLayer [40] used inter-layer pipelining in ReRAM-based accelerators, but did not consider dataflow optimizations. The fused-layer CNN accelerator sacrificed programmability to fuse the operations between layers in a fine-grained manner [2]. Li et al. implemented an inter-layer pipeline with an optimization for FC layers similar to ALLO [25]. But the design was limited to small CNNs and did not scale to large, complex networks such as ResNet and LSTMs. Shen et al. proposed to use heterogeneous engines to process different layers for higher utilization, but intermediate data were still written back to DRAM with no bandwidth and energy saving [38]. ScaleDeep was a scale-out server architecture that mapped entire NNs for training [44]. Its coarse-grained dataflow suffered from the inefficiencies as our baseline. To pipeline an NN across multiple GPUs, Jia et al. used dynamic programming to find the optimal strategies under a cost model [19, 20]. The above designs mostly mapped the whole NNs to the systems, and none of them studied the tradeoff of pipeline depth as TANGRAM does.

8 Conclusion

This work focused on dataflow that improves coarse-grained parallelism on tiled NN accelerators. We presented optimizations for both intra-layer parallelism and inter-layer pipelining that decrease buffer requirements, reduce off-chip accesses, alleviate frequent stalls in pipelined execution, and target complex DAG patterns in recent NNs. These optimizations provide significant performance and energy advantages over existing tiled and monolithic designs. Moreover, the benefits of these parallel dataflow optimizations will increase as NNs become larger and more complex.

Acknowledgments

The authors want to thank Zhihao Jia for the insightful discussion, and the anonymous reviewers for their valuable comments. This work was supported by the Stanford Platform Lab, NSF grant SHF-1408911, and SRC Center for Research on Intelligent Storage and Processing-in-memory (CRISP).

References

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *43rd International Symposium on Computer Architecture (ISCA)*. 1–13.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-Layer CNN Accelerators. In *49th International Symposium on Microarchitecture (MICRO)*. 22:1–22:12.

[3] Bradford M. Beckmann and David A. Wood. 2004. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 319–330.

[4] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 269–284.

[5] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *47th International Symposium on Microarchitecture (MICRO)*. 609–622.

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *43rd International Symposium on Computer Architecture (ISCA)*. 367–379.

[7] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 2016. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *2016 International Solid-State Circuits Conference (ISSCC)*. 262–263.

[8] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits (JSSC)* 52, 1, 127–138.

[9] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *43rd International Symposium on Computer Architecture (ISCA)*. 27–39.

[10] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. 2017. CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-circulant Weight Matrices. In *50th International Symposium on Microarchitecture (MICRO)*. 395–408.

[11] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *42nd International Symposium on Computer Architecture (ISCA)*. 92–104.

[12] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. Neuflow: A Runtime Reconfigurable Dataflow Processor for Vision. In *2011 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 109–116.

[13] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *45th International Symposium on Computer Architecture (ISCA)*. 1–14.

[14] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 751–764.

[15] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *43rd International Symposium on Computer Architecture (ISCA)*. 243–254.

[16] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *36th International Symposium on Computer Architecture (ISCA)*. 184–195.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[18] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4700–4708.

[19] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *35th International Conference on Machine Learning (ICML)*.

[20] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *2nd Conference on Systems and Machine Learning (SysML)*.

[21] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th International Symposium on Computer Architecture (ISCA)*. 1–12.

[22] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *43rd International Symposium on Computer Architecture (ISCA)*. 380–392.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *25th International Conference on Neural Information Processing Systems (NIPS)*. 1097–1105.

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015), 436–444.

[25] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A High Performance FPGA-based Accelerator for Large-Scale Convolutional Neural Networks. In *26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–9.

[26] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd International Symposium on Microarchitecture (MICRO)*. 469–480.

[27] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *43rd International Symposium on Computer Architecture (ISCA)*. 393–405.

[28] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A Flexible Dataflow Accelerator Architecture

for Convolutional Neural Networks. In *23rd International Symposium on High Performance Computer Architecture (HPCA)*. 553–564.

[29] Micron Technology Inc. 2007. TN-41-01: Calculating Memory System Power for DDR3. https://www.micron.com/support/tools-and-utilities/power-calc.

[30] Micron Technology Inc. 2014. Mobile LPDDR4 SDRAM: 272b: x64 Mobile LPDDR4 SDRAM Features.

[31] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *44th International Symposium on Computer Architecture*. 27–40.

[32] Maurice Peemen, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. 2013. Memory-Centric Accelerator Design for Convolutional Neural Networks. In *31st International Conference on Computer Design (ICCD)*. 13–19.

[33] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *43rd International Symposium on Computer Architecture (ISCA)*. 267–278.

[34] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19.

[35] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *40th International Symposium on Computer Architecture (ISCA)*. 475–486.

[36] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars. In *43rd International Symposium on Computer Architecture (ISCA)*. 14–26.

[37] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From High-level Deep Neural Models to FPGAs. In *49th International Symposium on Microarchitecture (MICRO)*. 17:1–17:12.

[38] Yongming Shen, Mechael Ferdman, and Peter Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *44th International Symposium on Computer Architecture (ISCA)*. 535–547.

[39] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (Sept 2014).

[40] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *23rd International Symposium on High Performance Computer Architecture (HPCA)*. 541–552.

[41] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *27th International Conference on Neural Information Processing Systems (NIPS)*. 3104–3112.

[42] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9.

[43] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Sotware-Defined Cache Hierarchies. In *44th International Symposium on Computer Architecture (ISCA)*. 652–665.

[44] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. 2017. ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *44th International Symposium on Computer Architecture (ISCA)*. 13–26.

[45] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2015. Show and Tell: A Neural Image Caption Generator. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 3156–3164.

[46] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. 2016. A Systematic Approach to Blocking Convolutional Neural Networks. *arXiv preprint arXiv:1606.04209* (Jun 2016).

[47] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In *44th International Symposium on Computer Architecture (ISCA)*. 548–560.

[48] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *23rd International Symposium on Field-Programmable Gate Arrays (FPGA)*. 161–170.

[49] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An Accelerator for Sparse Neural Networks. In *49th International Symposium on Microarchitecture (MICRO)*. 20:1–20:12.