PIM-DL: Boosting DNN Inference on Digital Processing In-Memory Architectures via Data Layout Optimizations

Minxuan Zhou*, Guoyang Chen†, Mohsen Imani‡, Saransh Gupta*, Weifeng Zhang† and Tajana Rosing*
*Department of Computer Science and Engineering, University of California, San Diego
Email: {miz087, sgupta, tajana}@ucsd.edu
†Alibaba Group
Email: {g.chen, weifeng.z}@alibaba-inc.com
‡Department of Computer Science, University of California, Irvine
Email: m.imani@uci.edu

***Abstract*—Digital processing in-memory (DPIM) provides very low overhead, highly parallel computation in conventional memory, which significantly accelerates data-intensive workloads like deep neural networks (DNNs). DPIM-based DNN accelerators require that data be properly laid out to make the best use of the available in-memory operations. However, existing DPIM accelerators tend to optimize for a particular DNN dataflow, neglecting the large design space of data layout. This work systematically investigates the data layout for DPIM DNN acceleration. We propose a mapping framework to represent the whole design space of DPIM data layout for general DNN models. Our investigation shows that an exhaustive exploration on the whole design space for mapping a DNN application to DPIM architecture is not computationally tractable. Therefore, we propose a compiler-level optimization, PIM-DL, that finds highly efficient data layouts for DPIM DNN acceleration using a two-level dynamic programming algorithm and a heuristic-based search. Our experiments show that DNN DPIM solutions created by our PIM-DL provide 3.7× and 4.3× better performance and energy efficiency as compared to the state of the art under the same hardware constraints.**

I. Introduction

Deep neural networks (DNNs) have been used in various application domains, ranging from natural language processing [1], [2], speech recognition [3], [4], to image object detection [5]–[9]. The size of emerging DNNs has become extremely large due to the need for high accuracy, posing significant challenges to conventional architectures because of limited parallelism and memory wall issues [1], [2], [10]–[13].

Processing In-Memory (PIM) is a promising non-conventional technology to accelerate emerging data-intensive applications by not only reducing the data movement but also increasing computing parallelism. Researchers have used PIM to accelerate neural networks [14]–[19] and other applications from a wide range of fields [13], [20]–[27]. Many PIM-based DNN accelerators rely on the analog domain of resistive memory (ReRAM) [14], [15], [19]. Although such accelerators provide significant improvement in both energy efficiency and performance, they are very inefficient area-wise due to costly peripherals needed for data conversion, have no floating-point support, and face scaling difficulty due to unstable multi-bit cells [14].

Digital PIM (DPIM), enables in-situ computations in conventional digital memory including ReRAM [28], [29], DRAM [30], [31], and SRAM [32], [33]. Since DPIM can directly work on digital data, it does not require costly peripherals for data conversion needed by analog PIM [14], [15], [19]. Several works have shown promising performance and scalability of DPIM-based DNN accelerators [16]–[18].

DPIM acceleration has strict requirements on how data should be placed in memory. Such data layout determines the degree of parallelism as well as other critical factors including memory utilization, and data throughput. The data layout problem in DPIM architectures is different from that in other types of spatial accelerators, which assume hierarchical architecture with separate storage components and processing elements. DPIM architecture only has a large digital memory which combines both computing and storage functionality. DPIM accelerators usually have thousands of large basic components (e.g., 1Mb digital memory block), leading to a large design space for optimization of application data and operation placement.

Most state-of-the-art DPIM DNN accelerators [16]–[18] use an output-parallel layout which allocates separate memory rows for computations of different output elements. Such layout requires significant data duplication because different outputs usually share a lot of input and filter data. As reported in NeuralCache [17], input loading and filter loading may contribute to 61% latency during the DNN inference. If we simply reorganize the layout to combine every two computations with a shared operand in the memory, we can approximately reduce 25% of data loading at a cost of double computation time. This shows there is a trade-off between computation parallelism and memory usage for DPIM data layout. There have been several work [34], [35] systematically investigated the design space of mapping one DNN layer to conventional DNN accelerators. However, this has not been done in the DPIM scenario. The design space of DPIM data layout becomes even larger for a

978-1-6654-4278-7/21/$31.00 ©2021 IEEE
DOI 10.1109/PACT52795.2021.00021

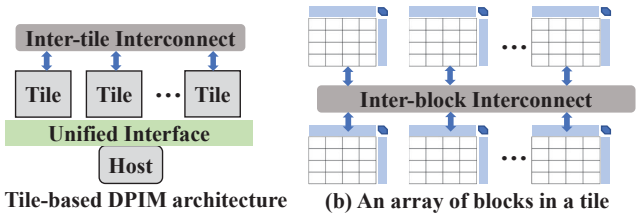
category of PIM-based DNN accelerators [14], [15], [18], [19], which pre-allocate memory resources for different program regions (usually layers). These accelerators, which we call static accelerators, introduce additional design dimensions for memory allocation. In this work, we show that the combination of conventional per-layer mapping and the DPIM-specific memory allocation forms an extremely large design space in DPIM architecture.

To fully explore the design space of DPIM DNN acceleration, we propose a new mapping framework of data layout. Unlike the state-of-the-art mapping framework [34], [35] for conventional DNN accelerators, the proposed framework considers both general and DPIM-specific dimensions of the design space. The decision of each dimension has an impact on multiple aspects of acceleration including computation cost, memory usage, and data movement pattern. All these aspects determine the overall efficiency of DPIM acceleration. With our framework, we can formulate the design space of data layout problem in DPIM DNN acceleration, which has an exponential complexity with the number of DNN layers and the number of available memory components (e.g., block) in the DPIM system.

The complexity of the design space makes it impossible for an exhaustive exploration using an efficient algorithm. Therefore, we further propose PIM-DL, a data layout optimization framework, to accelerate DPIM DNN inference. PIM-DL utilizes a two-level dynamic programming algorithm and a genetic algorithm based on heuristic search to efficiently find a good data layout based on application and hardware information, that performs better than previous methods. We evaluate PIM-DL on DPIM acceleration for several widely-used DNNs by an open-sourced DNN compiler and a cycle-accurate simulator. Our experiments show that PIM-DL can improve the performance of DPIM DNN acceleration under various architecture configurations by up to 1.9× while using 30% less memory without any hardware change. Furthermore, we apply PIM-DL to several state-of-the-art DPIM DNN accelerators and observe a 2.5× speedup on average. We also explore the design space of hardware-software co-design, inspired by our data layout experiments, and compare several customized systems with PIM-DL to state-of-the-art DPIM DNN accelerators varying in memory technologies and acceleration modes. Our experiment shows our software-hardware co-design systems provide 3.7× and 4.3× better performance and energy efficiency than corresponding baselines under the same hardware constraints.

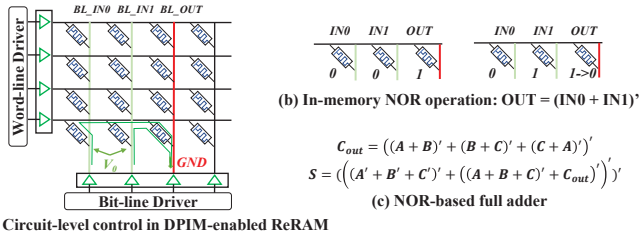
Overall, we make the following contributions in this work:

- This is the first work that comprehensively and systematically investigates the data layout problem in DPIM DNN acceleration. We generalize the data layout problem in DPIM using a DNN mapping framework for customized accelerators including DPIM-specific design dimensions which have not been investigated.
- We design compiler-level optimizations for a generic DPIM architecture to generate efficient data layout for DNN inference and dramatically boost performance and memory utilization of general DPIM DNN acceleration.
- Inspired by our experiments on data layout, we exploit the hardware-software co-design to build several customized systems which significantly improve the performance and energy efficiency of state-of-the-art.

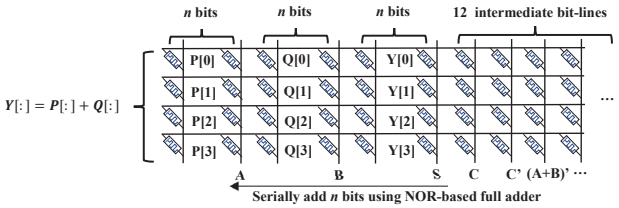


(a) Tile-based DPIM architecture (b) An array of blocks in a tile

Fig. 1. Generic DPIM architecture model.



(a) Circuit-level control in DPIM-enabled ReRAM (b) In-memory NOR operation: OUT = (IN0 + IN1)' (c) NOR-based full adder



(d) Bit-serial row-parallel vector addition

Fig. 2. Operations in a ReRAM-based DPIM.

II. Background and Motivation

In this section, we elaborate important preliminaries for DPIM-based DNN acceleration.

A. DPIM Architecture

This work targets a general DPIM architecture model which is shown in Figure 1. The DPIM architecture consists of multiple tiles, where tiles are connected to each other through an inter-tile network. Each tile contains several DPIM blocks and uses an inter-block interconnect to handle data movements between different blocks. We can configure this model to emulate state-of-the-art DPIM architectures by customizing hardware characteristics like operation latency and memory structure.

Several recent works have implemented DPIM functionality in various digital memory technologies including SRAM [17], [32], DRAM [16], [31], and ReRAM [18], [28], [36]. The common backbone of various DPIM technologies is row-parallel bit-serial operation, which exploits the shared bit-line circuits in digital memory block to process all rows in a bit-line using a single step. When using an universal bit operation,

DPIM-enabled memory can support custom computations by sequentially executing multiple bit-line steps. We take an example of DPIM-enabled ReRAM block to illustrate basic DPIM operations. Figure 2(a) shows a memory block which includes a crossbar of ReRAM cells and peripheral circuits for driving bit-lines and word-lines. By applying specific voltage to the bit-lines, the value stored in a resistive cell may change. This operation can take effect on all word-lines (rows) enabling highly parallel operations. Figure 2(b) shows an example of computing NOR operation, a universal operation that can be used to implement custom functions. Figure 2(c) shows a NOR-based 1-bit full adder that takes 12 NOR steps.

We can exploit this 1-bit full adder to support multi-bit operations. Figure 2(d) shows an example of addition for two vectors with 4 n-bit values. The memory sequentially applies the full addition to each bit of computations by reusing the carrier. In addition to the carriers, we also need to reserve several bit-lines (12 in this case) to store intermediate values like $(A + B)'$. These intermediate bit-lines can be shared across computations. This scheme can process an n-bit vector addition in $12n + 1$ steps for all elements. When processing long vectors, this brings a significant performance benefit because of the extremely high degree of parallelism. Other than addition, we can also implement different custom functions including multiplication, subtraction, and division. In addition to integer values, DPIM also supports floating point by separately computing exponent and sign bits [18].

We should note that different memory technologies have different schemes for DPIM operations. For example, ComputeCache [32] and NeuralCache [17] proposed to exploit the sense amplifier in SRAM to sense shared bit-lines between two activated rows. ComputeDRAM [31] utilized specialized sequences of DRAM commands (e.g., row activation and pre-charge) to implement DPIM operations in commodity DRAM chips by only slightly modifying the memory controller. The high-level computing scheme of different DPIM technologies is still row-parallel bit-serial operation, so that we can design several general strategies, including the data layout, for most DPIM accelerators. We refer readers to previous works for more details on the circuit-level design and implementation for different DPIM technologies [16]–[18], [31], [32].

B. DPIM-based DNN Accelerators

DPIM architecture can emulate a large group of SIMD processing units, which is promising to accelerate DNN applications. We can categorize DPIM DNN accelerators into two groups, dynamic and static, based on the acceleration mode as shown in Figure 3(a). On the one hand, dynamic DPIM DNN accelerators allocate all memory resources to process one DNN layer at a time. The weights and inputs of the layer need to be loaded to the memory before computation. After the layer is done with computation, its results are reorganized in the memory to compute the next layer. Even though the dynamic accelerator fully utilizes memory resource for computations, it has a couple of drawbacks. First, weight loading for each layer

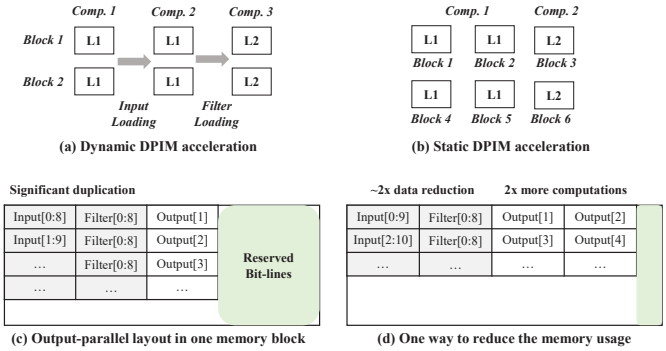
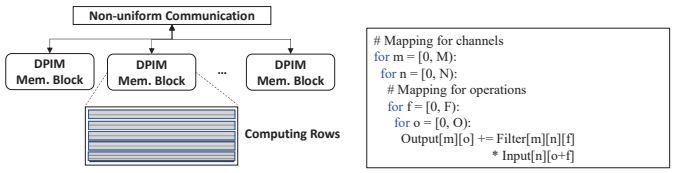


Fig. 3. DPIM DNN accelerators and data layout.

would consume significant amount of time and energy due to data loading from external memory [17]. Second, processing one layer at a time cannot support pipelined execution between different layers for more throughput. On the other hand, the static acceleration mode stores weights for the whole DNN model statically and has been widely adopted in many types of PIM DNN accelerations, like analog PIM (e.g., Pipelayer [19], ISAAC [14], Prime [15], etc.) and near data processing (e.g., Tetris [37]). In the static acceleration, a portion of the memory handles computations for a specific layer, and sends results to another portion of the memory handling the next layer. Even though static accelerators require more memory, it avoids weight loading during the runtime and can pipeline the execution of different layers to improve the throughput. Since dynamic acceleration can be represented as a special case of static acceleration, this work focus on the static DPIM acceleration and provide results to both dynamic and static acceleration in Section VI.

State-of-the-art DPIM DNN accelerators, both dynamic and static, use the fully output-parallel mapping for all DNN workloads. Figure 3(c) shows an example of 1D convolution with a 1*9 filter using the fully output-parallel layout used by NeuralCache [17], which allocates the computation for each output element in a memory row, exploiting row-parallel operations to process all outputs in parallel. However, such output-parallel mapping introduces data duplication that may significantly increase the memory usage. In dynamic accelerators, large memory usage may require the system loads data several times for a layer since that cannot fit all data. This introduces large overheads due to data loading and computing sub-parts sequentially. For static accelerators, this data layout may significantly increase the memory requirement for DNN workloads. To reduce the memory usage, Figure 3(d) shows one method that combines each two computations, sharing an operand, to one memory row. This method reduces the memory usage by almost $2\times$ at the cost of $2\times$ more row-parallel operations. There are other design dimensions that we can fine-tune the data layout to achieve better performance by comprehensive exploration.

Recent DNN mapping frameworks, such as Timeloop [35] and Interstellar [34], explore a relatively limited design space



(a) DPIM accelerator model. (b) An example 1D-Conv layer for mapping.

Fig. 4. (a) The architectural abstraction of DPIM accelerator using conventional DNN accelerator model [34], [35]. (b) An example of 1D convolution layer.

of DNN application mapping, with focus on a basic nested loops and primitives such as loop splitting and loop reordering. We refer to them as conventional frameworks. Even though the DPIM architecture is similar to single-level distributed processors (e.g., memory rows), the conventional frameworks cannot fully explore the design of data layout in DPIM architectures for three reasons. First, the DPIM architecture has a single type of component that acts as both a compute and a storage unit. However, the conventional framework does not consider the data layout for different computation mappings. Second, the conventional framework only explores the design space of a single layer, which cannot support static DPIM accelerators. DPIM accelerators require a holistic framework that consider the global constrains of the architecture when mapping all the layers. Third, the locations of the memory partitions allocated to different layers also have an impact on the overall system performance because of the inter-layer data movements. For example, we may need to move the output of the preceding layer to the input of the succeeding layer. Most accelerators have a non-uniform communication network for all processing units (e.g., memory blocks or rows in DPIM architectures), so the overhead of data movement varies as a function of the allocation scheme. In this work, we propose a novel mapping framework for DNN data layout on DPIM architecture (Section III), and an efficient data layout optimization that finds a good data layout in the large design space (Section IV).

III. PIM-DL Data Layout Framework

We formulate the data layout of a DNN model with n layers in DPIM architectures using two sets: global layout strategy $L = \{l_i, i \in \{1...n\}\}$, and memory allocation $M = \{m_i, i \in \{1...n\}\}$. Specifically, l_i is the data layout strategy for layer i. In Section III, we introduce a way to use the conventional DNN mapping framework to represent the layer layout in DPIM. Furthermore, m_i is a set of memory resources allocated for layer i using the layout l_i. As compared to the conventional framework, the data layout of DNN model in DPIM architecture has a significantly larger design space. Assuming the size of design space of conventional framework is S, the design space of L has a size of S^n. For each global layout strategy L, the number of possible memory allocations is $N_m(N_m-1)...(N_m-k+1)$, where N_m is the number of memory resources in the DPIM architecture and k is the number of memory resources required for L. In this work, we use the memory block as the granularity of memory resource allocation.

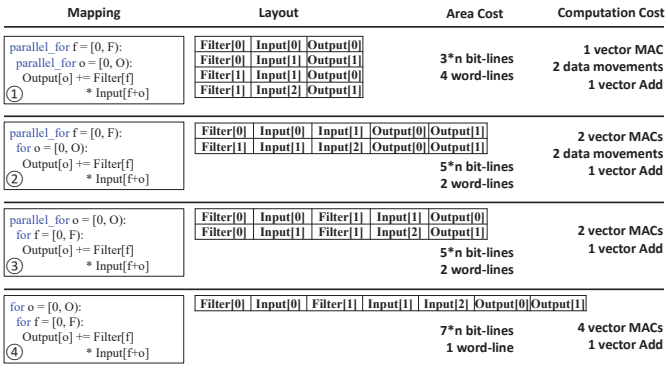


Fig. 5. Data layouts of different 1D-Conv mappings.

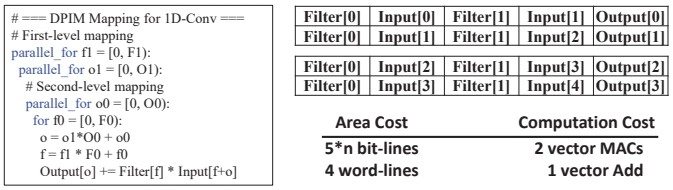


Fig. 6. Loop tiling with an additional level.

Our mapping framework for DNN applications enables optimization across the full design space of DNN data layout on DPIM architectures. This section illustrates all design dimensions of the framework which impact different aspects of DPIM DNN acceleration including computing parallelism, memory utilization, and data transfer pattern. We investigate data layouts and corresponding cost models of different DNN mappings through the example 1D-conv layer as shown in Figure 4(b). The example convolution has N input channels and M output channels. The filter size is F and the convolution generate O outputs for each output channel. We select convolution as the main example because it is the most complex and time-consuming in a wide range of emerging DNNs. We should note that our analysis is applicable to other layers like fully-connected layer, which can also be represented as a nested loop.

A. Operation Layout

We first investigate the detailed data layout of mapping a single 1D convolution. Figure 5 shows layouts and cost models of 4 basic mappings. We use $parallel\_for$, used in Timeloop [35], to indicate a spatial parallel for a loop in the mapping. As mentioned before, the basic hardware components for resource allocation is memory rows, so that a $parallel\_for$ places all operands for each item in a row and aligns computations for all items in the same bit-lines. In this case, DPIM can process all these items in parallel using row-parallel bit-serial operations. In addition, a normal for places computations for items in a row, requiring sequential execution. The most parallel mapping is Mapping 1 (Figure 5) which

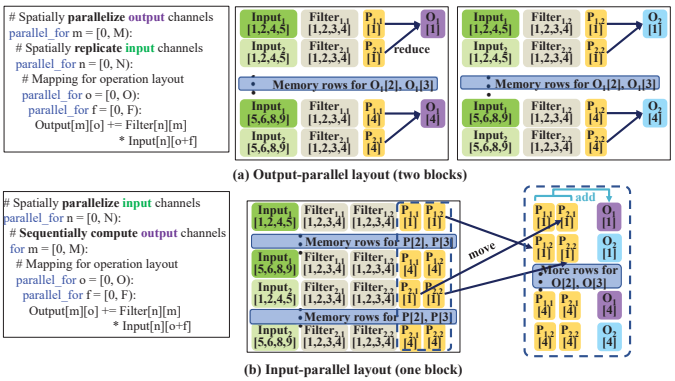


Fig. 7. Two channel-parallel data layout schemes for convolutions of two input/output channels.

parallelizes all computations. However, such straightforward layout requires data duplication to maximize the parallelism. Furthermore, Mapping-1 needs to re-align partial sums to generate final outputs, requiring extra data movements.

One way to reduce the data size is to combine rows with shared elements by folding the loop. For example, Mapping-2 folds computations of 2 outputs in one row for each filter replica, saving memory space for 2 elements. However, such mappings introduce extra vector MACs. We can also avoid data movements by changing the order of the loop as shown in Mapping-3 and Mapping-4. We can further enlarge the design space by add another level of loop in the operation, as shown in Figure 6. The example shows we can control the number of parallel computation outputs by adopting different parallel schemes in two levels. According to the data size and architecture configurations, the most efficient mapping might be different. Furthermore, the memory requirement (including the dimension of bit-lines and word-lines) is also important since it influences the constraints for other design dimensions.

B. Layer Layout

Emerging DNNs usually apply operations on multi-channel data to capture comprehensive features. The result of each output channel is a function of different input channels with corresponding filters. Such channel-level parallelism provides another dimension of the design space for data layout. In the conventional mapping framework, we can explore the layer layout by changing the order of loop. Figure 7 shows two example mappings of layer layout to illustrate the cost models in the layer-level.

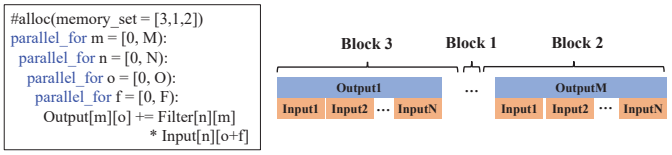
1) Output-parallel Data Layout: We can parallelize a convolution layer along the dimension of output channel because computations of output channels are independent. Figure 7(a) shows an output-parallel mapping which schedules a $parallel\_for$ for the output channel in the outer loop. This example mapping further apply $parallel\_for$ for all inner loops to fully parallelize convolutions for all output elements. Because of such spatial parallelism, we need extra data movements to reduce partial sums to generate final outputs. For example, we align $Input_1[1,2,4,5]$ and $Input_2[1,2,4,5]$ in the first two memory rows in Figure 7(a), and compute partial results of $O_1[1]$ by convolution with $Filter_{1,1}$ and $Filter_{2,1}$ respectively. We should reorganize partial results of each output element in the memory to compute the output element by additions. We can take $log_2 N$ steps to complete such reduction, by moving and adding half of partial sums at each step. Reduction in each memory block can happen in parallel because a typical memory block (e.g., 1K rows) can fit all convolution data for an output element which requires N rows where N is less than 1024 for most current CNNs.

Such output-parallel layout achieves high parallelism while it may require too many memory blocks to fit $N*M*O$ rows. To solve such issues, we can adopt more sequential operations for the operation layout to reduce the number of rows by combining multiple convolutions in the same row. In this case, we can change the inner-loops for operation layout to fold computations inside each channel. Furthermore, we can break the channel-level loops into more levels and fold computations across channels. Such computation folding comes at the cost of using more bit-lines and sacrificing parallelism.

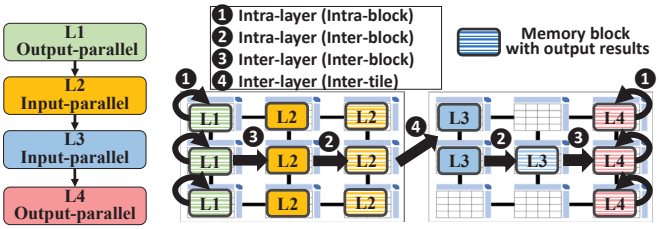
2) Input-parallel Data Layout: Another way to parallelize a convolution layer is to schedule computations of different input channels in parallel. DPIM architecture can implement this strategy by aligning the computations in different input channels for a specific output element, as shown in Figure 7(b). In the input-parallel layout, computations of each input channel generate partial results that need to be summed up with partial results in other input channels to calculate the output results. Since partial results for a specific output channel distribute vertically, we cannot accumulate them directly by PIM operations, requiring data movements to realign them (right part of Figure 7(b)).

The input-parallel layout requires less number of word-lines (rows) than the output-parallel layout - $N*O$ as compared to $N*M*O$. However, it requires a large number of bit-lines, which is equal to $M*F*b$, to fit all M filters for an input channel to compute partial results in b-bit precision. For example, in a common 3*3 convolution with 8-bit fixed point values, each filter requires 72 bit-lines in the memory; this means a memory block with 1024 bit-lines can only fit 13 filters. If the number of output channels is larger than 13, we have to distribute all filters for an input channel across multiple blocks, causing extra inter-block data movements during the sequential execution. A trade-off we can adopt to improve the parallelism is storing a copy of input in each block so that all blocks can process convolutions and reductions in parallel. Similar to output-parallel layout, we can fine-tune the input parallel layout by changing the loop structure.

3) Other Design Dimensions: The basic difference between output-parallel and input-parallel layout is the order of loops for input and output channels. As introduced before, we can break these two loops into more loop levels to change the data layout in DPIM. Two examples shown in Figure 7 assume loops of operation mapping always in the inner loops. With the conventional framework, it is possible to explore more mappings by shuffling the order of these four loops



(a) Resource allocation for 1 layer with *memory_set*.



(b) Global layout for 4 layers with different layer layout strategies.

Fig. 8. Memory allocation and global layout.

which can change the order of partial sums aligned in the memory. However, the cost model of layout layer depends on the relative order between input-channel and output-channel loops which determines the layout for partial sums. Therefore, we can customize any layer layout based on the two basic mappings introduced in this section.

C. Memory Allocation

After determining the detailed layout for one layer, the next step is to allocate memory resource (e.g., memory blocks) for the layer. Considering the DPIM system usually contains a large number of memory blocks, the memory allocation for a layer also adds a dimension in the design space which indicates the memory blocks we want to allocate for a layer with a given layout. Therefore, we add an *alloc* configuration in the DPIM mapping framework which gets in an ordered set of memory blocks ($memory\_set$), as shown in Figure 8(a). With the detailed layer layout, we evenly divide all parallel computations to the ordered memory blocks. The figure shows an example of allocating three blocks in the order of (3,1,2) to a layer using output-parallel layout. Based on the mapping shown in the left side of Figure 8(a), the layout parallelizes computations of all output channels, where each output channel parallelizes computations using all input channels as shown in the right side of Figure 8(a). Inside each output-input computation, the detailed operation layout is defined by the two inner loops in the mapping. Then, the memory allocation for set (3,1,2) evenly divides these computations into three segments, and allocate the first, the second, and the third segment to block 3, 1, 2 respectively.

In static DPIM DNN accelerators, we need to distribute layers across the global memory, introducing the problem of global layout. The global layout can be represented by allocating different sets of memory to all layers. The number of required blocks for each layout depends on its layout strategy (including both operation and layer layout). The global layout introduces various data movements. The first type of data movement is intra-layer, which happens during the layer computation. Both channel-parallel data layouts would cause intra-layer data movements when accumulating partial results. As analyzed in Section III-B, output-parallel layout accumulates partial sums in the same memory block since it aligns all input data for a specific output element in consecutive rows; and input-parallel layout reserves a specific set of blocks to reduce all partial results distributed across different blocks. As shown in Figure 8, intra-layer movements of output-parallel layers (L1 and L4) happen inside each block, while those of input-parallel layers (L2 and L3) use the inter-block interconnect. Other than intra-layer data movements, data dependency also happens between different layers. Such dependency results in inter-layer data movements which may use inter-block interconnect or inter-tile interconnect.

Algorithm 1: Tile-level optimization

Data: $Layers[N]$
Result: $Layout[N][N]$
Function *optimizeLayerGroup(group: layer[n], nBlk: int)*
  $f[:N][:N] = inf;\ f[0,:N] = 0;$
  $decision[:N,:N] = null;$
  for $i \leftarrow 1$ **to** n **do**
    foreach *layout* t *for* $group[i]$ **do**
      $rb = MemoryCost(i,t);$
      for $j \leftarrow req\_blk$ **to** $nBlk$ **do**
        if $f[i][j] > f[i-1][j-rb] + PerfCost(i,t)$ **then**
          $f[i][j] = f[i-1][j-rb] + PerfCost(i,t);$
          $decision[i][j] = t;$
/*Generate the optimized layout for each layer group based on $decision$*/

To ensure the generality of memory allocation, we can define any memory set with blocks that have not been allocated for other layers. However, allocating memory blocks that have long distance with each other (e.g., in different memory tiles) to a layer may introduce large data movement overhead if the layer layout requires inter-block data transfer for some operations (e.g., reduction). Furthermore, allocating long distance memory blocks to layers with data-dependency will also cause large data movement overhead. Since different data movement patterns take various latency (e.g., inter-tile movements are usually slower than inter-block movements), it is important to carefully design the global layout based on both the DNN structure and the architecture configuration. In Section IV, we propose a holistic data layout optimization for general DPIM DNN acceleration.

IV. PIM-DL Optimization

The PIM-DL framework shows DPIM DNN acceleration has a large design space which is usually too large to be exhaustively searched for an optimal data layout. In this section, we introduce an optimization algorithm which holistically optimizes data layout for general DPIM DNN acceleration. PIM-DL optimization includes three steps, tile-level optimization, global optimization, and block allocation, to efficiently find an efficient data layout.

A. Tile-level Optimization

The goal of the tile-level optimization is to find the optimized data layout for allocating a `layer group` in a tile, where a `layer group` denotes a group of DNN layers allocated to the same tile. However, the number of possible combinations of DNN layers is too large to be efficiently explored. Therefore, we limit the exploration of single-tile data layout to consecutive DNN layers in the network based on the fact that the bandwidth of intra-tile data bus is much higher than the inter-tile interconnect, making it more efficient to process consecutive layers in a tile. The tile-level optimization searches for the best layout strategy for each set of consecutive layers that fit them in the same tile.

Algorithm 1 outlines the steps of optimizing a layer group with consecutive DNN layers, which is based on dynamic programming [38]. Each state $f[i][j]$ stores the minimum cost of allocating the first i DNN layers using j memory blocks. To calculate each $f[i][j]$, the algorithm considers all possible data layout strategies for each layer by exploring the PIM-DL data layout framework introduced in Section III. For each layout strategy t, we can calculate the number of required memory blocks (rb). We can estimate the cost for adopting layout t for layer i by adding $f[i-1][j-rb]$ and the estimated performance cost of the layout, which includes data loading, computation, intra-layer data movement, and inter-layer data movement.

The data loading cost and the computation cost can be accurately calculated for each layer with a specific layout. These costs mainly change as a function of data dimensions of the DNN layer. For the intra-layer data movement, we cannot directly calculate it because we do not know which memory blocks we allocate to each layer at this point. We approximate this cost by assuming the system has a uniform data bus (e.g., shared bus or fully-connected interconnect) inside each tile so that the intra-layer data movement cost is a function as the size of moved data. In Section IV-C, we introduce a way to reduce the estimation error for intra-layer data movement cost by a block allocation method which uses a genetic algorithm to find an efficient block allocation for a specific layer which has a similar intra-layer data movement cost as our ideal assumption. The inter-layer data movement cost is calculated based on the average bandwidth of inter-block network if the input data comes from a layer in the same layer group; otherwise, it is calculated based on the average bandwidth of inter-tile network. Our cost estimation is based on hardware simulation, which is introduced in Section V.

The dynamic programming algorithm runs with all layer groups of consecutive DNN layers ($N(N+1)/2$ in total) and generates the optimized layout strategy for each possible layer group. The exploration results are used by the global optimization algorithm which finds the best allocation for the whole network in the multi-tile architecture.

B. Global Optimization

The tile-level optimization finds the optimized layout for a specific layer group in a single tile, which can give the optimal layout if the whole DNN can fit in a tile. However, current DNNs for real-world applications are usually large and deep, which may require multiple tiles in static DPIM accelerators. Therefore, we propose a cost-aware algorithm for dividing a large DNN into multiple tiles while minimizing the overall cost considering inter-tile data movements. Algorithm 2 shows the process of global optimization which is also based on dynamic programming. Each state $f[t][i]$ denotes the minimum cost of allocating layer $0-i$ in t tiles. The algorithm calculates the minimum cost from a single tile to nT tiles. For each tile t, the algorithm iterates over all layers in a topologically sorted order to calculate $f[t][i]$. For each $f[t][i]$, the algorithm checks all possible continuous layer group $j-i$ and updates $f[t][i]$ if $f[t-1][j-1]+cost(j,i)$ is less than $f[t][i]$. The $cost(j,i)$ is the minimized cost of allocating layer j to i in a tile, which has been calculated in the tile-level optimization phase. We record layout decisions during the execution, and generate the best layout we found by backtracking from $f[nT][N]$.

Algorithm 2: Global optimization

Data: $Layout[N][N], tiles[nT]$
Result: $TileLayout[nT]$
$f[0:nT][0:N] = inf$; $f[0][0] = 0$;
$decision[0:nT][0:N] = null$;
for $t \leftarrow 1$ **to** nT **do**
  for $i \leftarrow 1$ **to** N **do**
    for $j \leftarrow 0$ **to** $i-1$ **do**
      if $f[t][i] > f[t-1][j] + Cost(Layout[j+1][i])$ **then**
        $f[t][i] = f[t-1][j] + Cost(Layout[j+1][i])$;
        $decision[t][i] = j$;
/*Generate tile allocation for all layers based on $decision$*/;

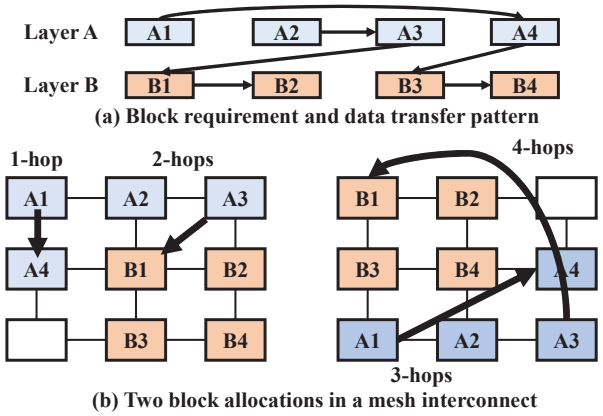


Fig. 9. Block allocations in a mesh interconnect.

C. Block Allocation

After global layout optimization, we generate data layout strategy and tile allocation for all DNN layers. However, the previous two steps do not consider locations of blocks allocated to each layer (*memory_set*). For an architecture with a uniform-latency data transfer network in each tile (e.g., shared bus), the block allocation would not impact the overall performance since all inter-block data transfers have the same latency. However, DNN accelerators usually have

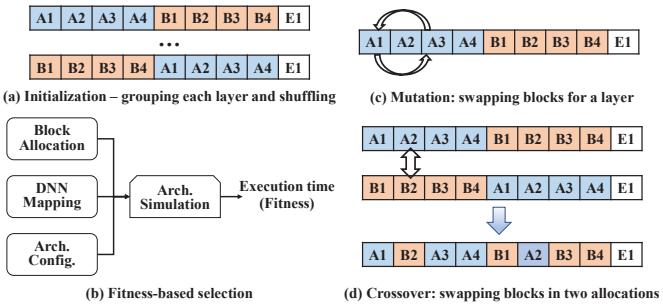


Fig. 10. The genetic algorithm for block allocation.

customized data links with non-uniform latency. For example, FloatPIM [18] adopts a latch-based linear data link which can efficiently handle data movements between consecutive DNN layers. In architectures that use non-uniform data links, different block allocations may exhibit very different performance results. Figure 9 shows two block allocation schemes for two connected layers, each of which requires 4 blocks based on the layout. We assume a simple mesh network for inter-block connection. The figure shows the data transfer from A1 to A4 can be directly handled in the mesh interconnect for the first allocation, while requiring 3 hops in the second one.

For a given interconnect structure, the only way to find the optimal block allocation for a layer group is exhaustive search which is not computational tractable since a tile usually contains hundreds of memory blocks. In order to generate efficient block allocation for general DPIM architectures, we use a heuristic search based on genetic algorithm [39], as shown in Figure 10. Specifically, we first encode all blocks allocated to each layer and use a vector as the genetic representation. If a tile is not fully used, we also encode empty blocks (e.g., E1 in Figure 10) to keep the length of vector the same as the number of blocks in a tile. During the initialization (Figure 10(a)), we first generate allocations that group blocks for each layer together and shuffle the order between different layers. For each generation, we define the fitness as the execution time based on hardware simulation (Figure 10(b)). Specifically, our simulator estimates the execution time by taking in the block allocation, the layout strategy, and the hardware configuration. For each generation, the mutation operation randomly swaps two blocks in the same layer and the crossover operation swaps the same position in two allocations (Figure 10(c)). We run the genetic algorithm for 3000 generations to find the near-optimal block allocation within reasonable time.

V. Methodology

Compiler Implementation. We implement PIM-DL in Glow, an open-source machine learning compiler for heterogeneous architectures [40]. We instrument the graph lowering engine of Glow for the data layout optimization. We add a new back-end of DPIM in Glow, which generates DPIM operation trace based on the optimized data layout. We then use an in-house cycle-accurate simulator for evaluation.

TABLE I
Hardware Parameters for ReRAM Device.

k_{on}	k_{off}	$\alpha_{on}, \alpha_{off}$	$V_{T,ON}$	$V_{T,OFF}$	x_{on}
$-216.2m/s$	$0.091m/s$	4	$-1.5V$	$0.3V$	0
x_{off}	R_{ON}	R_{OFF}	E_{set}	E_{reset}	E_{NOR}
$3nm$	$10K\Omega$	$10M\Omega$	$23.8fJ$	$0.32fJ$	$0.29fJ$
E_{search}	T_{NOR}	T_{search}	T_{Switch}	V_{RESET}	V_{SET}
$5.34pJ$	$1.1ns$	$1.5ns$	$1ns$	$1V$	$2V$

TABLE II
Architectural Parameters.

Memory Block		
Organization	#bit-lines (columns)	1024
	#word-lines (rows)	1024
Tile		
Block array	#blocks	256
	Dimension	16*16 by default
DPIM System		
Organization	#tiles	32
	Dimension	1*32 linked by chain
Serial links (Inter-tile)	Bandwidth	160GB/s
	Latency	8-cycle

DPIM Simulation. Our simulation adopts a two-step method which has been widely used in several previous works on emerging architectures [14], [19], [41]. We first model timing and energy parameters for operations on different hardware components using validated circuit-level simulators; and then use these numbers in the simulator to estimate the performance of different architecture configurations. We investigate three widely used memory technologies for DPIM acceleration including ReRAM [18], DRAM [16], and SRAM [17], [32].

The basic ReRAM technology used in this work is the Voltage ThrEshold Adaptive Memristor (VTEAM) model [42] with I_{ON}/I_{OFF} ratio of 10^3. The detailed parameters, including both energy and timing, of the VTEAM model is listed in Table I. We use HSPICE design tool for circuit level simulations to provide timing and energy results for ReRAM operations. The DRAM specification used in this work is extracted from a published datasheet from the industry [43]. We model all CMOS components (including buffers and interconnects) in Cacti [44] at 32nm technology. The interconnect is modeled by Orion 3.0 [45] in 45nm technology, and we scale the results to 32nm technology.

Hardware Configurations. Our experiments cover a wide range of hardware configurations, which will be detailed in corresponding sections. However, the high-level organization of tile-based architecture is shown in Table II, which has 32 tiles, and each tile has 256 memory blocks. Each memory block contains 1024 bit-lines and 1024 word-lines, providing a total size of 8Gb. We investigate three different memory technologies in the baseline architecture model, and show the cross-technology results in Section VI-C. We model the on-chip NoC (inter-block interconnect) with 128-bit channels and assume 3 cycles for router and 1 cycle for wire as the zero-load delay [37]. We simulate different structures for inter-block NoC and show the performance comparison in Section VI-A3. The inter-tile network is modeled as SerDes link used by HMC

TABLE III
TESTED DNN MODELS.

Network	Depth	Width	#MACs	#Para.
AlexNet [5]	7	1	7.27G	60.97M
DenseNet [8]	120	1	4.87G	25.56M
GoogleNet [7]	21	4	16.04G	7M
InceptionV2 [47]	33	4	12.27G	72.56M
MobileNet [48]	53	1	573.78M	4.23M
ResNet50 [6]	49	2	3.87G	46.72M
Vgg19 [49]	18	1	196.32G	314.12M
SqueezeNet [50]	17	2	861.34M	12.58M

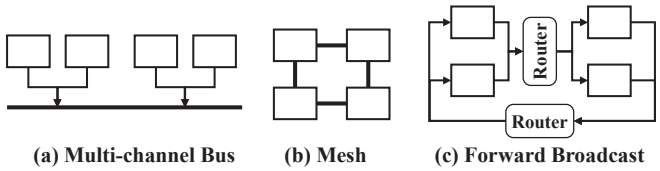


Fig. 11. Interconnect structures used in our experiments.

with an average 160GB/s bandwidth [46].

DNN Workloads. We use 8 popular DNN models in our experiments, as shown in Table III, including AlexNet [5], DenseNet [8], GoogleNet [7], InceptionV2 [47], MobileNet [48], ResNet50 [6], Vgg19 [49], and SqueezeNet [50]. We test all models on the inference task for ImageNet dataset [51].

VI. EXPERIMENTS

A. Data Layout Optimization

To verify the efficiency of PIM-DL, we compare it to several heuristic-based methods which adopt a fixed strategy for all DNN layers. Furthermore, we conduct such experiment on various architecture configurations to justify the generality of PIM-DL.

1) Software and Hardware Baselines: All baseline layout strategies are fully parallel for either input-level or output-level. Such strategies are commonly used in previous DPIM DNN accelerators including FloatPIM [18], Drisa [16], NeuralCache [17]. We denote input-parallel and output-parallel schemes as `In` and `Out` in all figures. For each channel parallel method, we show results of three fine-tuned mappings by selecting different degrees of parallelism, denoted as `Max`, `Mid`, and `Min`. We adopt a sequential block allocation for all baselines, where blocks allocated to each layer are placed sequentially in the memory.

All baseline architectures are ReRAM-based static accelerators with different interconnect networks as shown in Figure 11. We test 1 uniform interconnect, `Bus`, indicating a global bus shared by all blocks in a tile. Each 8 blocks in a tile share a channel, giving a 512 GB/s total bandwidth which is similar to a HMC chip with 32 vaults [46].

We also test 2 non-uniform interconnects: `Mesh` and `Broadcast` as shown in Figure 11. Specifically, `Mesh` is a

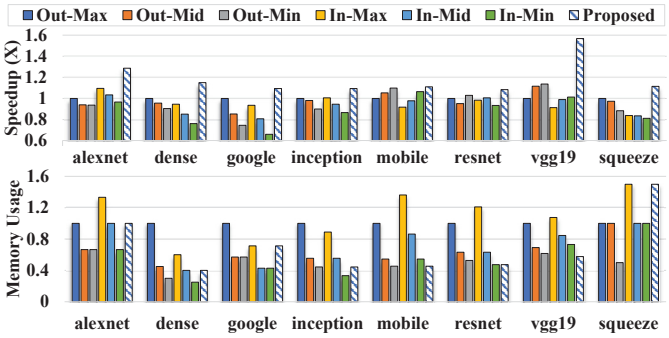


Fig. 12. Performance and memory usage results of heuristic-based layout schemes and PIM-DL with a `Bus` interconnect for blocks in a tile. The results are normalized to `Out-Max` layout.

widely used interconnect in domain-specific accelerators [52], [53]. `Broadcast` is a modified version of interconnect used in FloatPIM [18]. The original interconnect organizes all blocks in a chain to fit the sequential data transfer pattern of DNN workloads. However, FloatPIM assumes each block can process one layer which is different from the generic DPIM acceleration as analyzed in Section III. In our experiments, a DNN layer usually requires multiple blocks, leading to a broadcast data transfer pattern. The `Broadcast` network organizes blocks in different column and support single-direction 1-to-N data transfer between two columns. We group 16 blocks in a column so that each tile has 16 block columns.

For the inter-tile interconnect, all baselines assume a Mesh-like network in these experiments because our experiments show that most widely used inter-tile connection networks, including Mesh, Chain, and Bus, give similar results because data transfer pattern between tiles is simple.

2) Comparison on Uniform Interconnect: To evaluate the efficiency of different parts in our optimization, we show the performance and memory usage on `Bus` (Figure 12) to exclude the effect of block allocation optimization. As shown in the results, the average speedups provided by the data layout optimization is 18.8% as compared to the output-parallel layout used in state-of-the-arts (`Out-Max`). Across different tested DNNs, the data layout found by our optimization framework can improve the performance by 41.0% and 13.6% as compared to the worst and the best heuristic-based methods on average, respectively. The results show that a single heuristic-based data layout cannot provide the optimal performance for all scenarios. Furthermore, the optimization decreases 30.5% memory usage of `Out-Max`. The results show that PIM-DL can always provide the best performance, while heuristics without adaptive layout strategies lead to sub-optimal performance and memory utilization.

3) Comparison on Non-Uniform Structures: We then compare the data layout optimization with heuristic-based methods on different interconnect structures. Figure 13 shows the performance results of `Out-Max` and the optimized layout found by our framework on three interconnect structures. All results are normalized to *Bus Out-Max*.

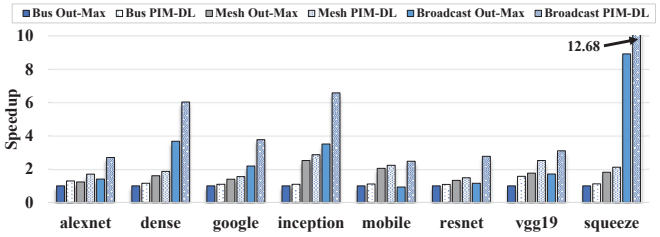


Fig. 13. The performance of `Out-Max` layout and the optimized layout across interconnect structures.

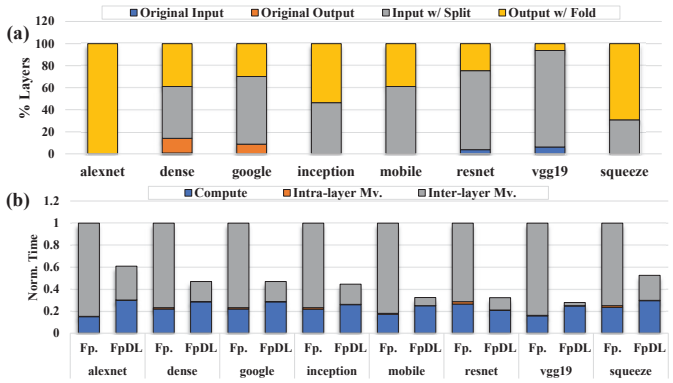


Fig. 14. (a) The percentage of layers using different layout schemes on FloatPIM [18] with our optimization; (b) Normalized performance breakdown of original FloatPIM (Fp) and FloatPIM with PIM-DL.

We first investigate the performance improvement provided by data layout optimization on non-uniform interconnects. As compared to the fully output-parallel layout, data layout optimization can provide 1.20× and 1.94× speedup on `Mesh` and `Broadcast` respectively. Such results indicate that PIM-DL can improve the performance of DPIM DNN acceleration across a wide range of architectures. Furthermore, PIM-DL provides more speedup on customized architectures.

Such experiment results also show a significant benefit provided by interconnect customization. With the data layout optimization, `Broadcast` is 1.61× faster than `Mesh`. Furthermore, `Broadcast` interconnect requires 81.1% less area as compared to `Mesh` interconnect. Such improvements on area efficiency come from significantly less routers, even though each router takes larger area because of large multiplexer.

B. Applicability to other DPIM Accelerators

We apply PIM-DL to previous DPIM accelerators by modeling costs of different operations based on the specific architecture design. We first utilize PIM-DL on FloatPIM [18] as an example. Figure14(a) shows the percentage of layers using different data layout schemes decided by the optimization. The result shows that over 95% of layers can improve the performance by using fine-tuned strategies. We should note that only 1% layers keep using the original scheme of FloatPIM (fully output-parallel). Figure 14(b) shows the normalized time

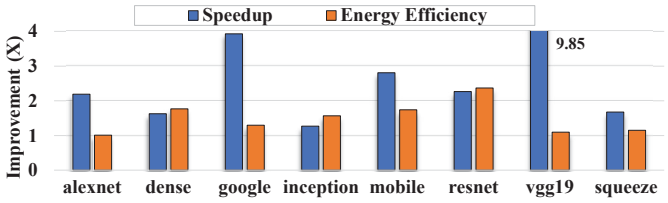


Fig. 15. Performance and energy improvements on NeuralCache [17].

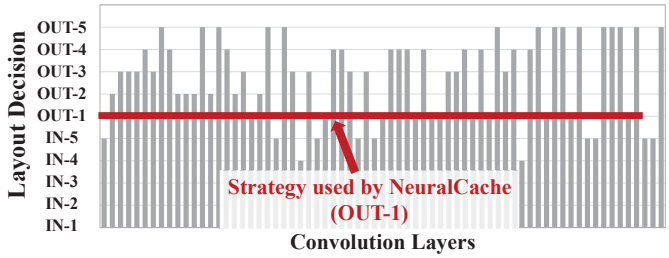


Fig. 16. PIM-DL data layout of InceptionV2 on NeuralCache [17].

breakdown of the original FloatPIM and our optimization. The result shows that the optimization can significantly reduce the data movement overhead, leading to a 2.5× speedup over the original FloatPIM.

Even though PIM-DL is mainly designed for DPIM DNN accelerators using static computing model, we can still achieve performance improvement by data layout optimization in dynamic DPIM accelerators like NeuralCache [17]. Since the dynamic DNN acceleration exploits the whole memory for each layer, we can still explore different mappings to find the most efficient layout for each layer. Figure 15 shows the performance and energy efficiency improvements provided by our exploration, which are 2.6× and 1.5× respectively.

Figure 16 shows detailed layout strategies for all layers in InceptionV2 determined by our optimization. Similar to previous experiments, `In` and `Out` indicate input-parallel and output-parallel layouts respectively. The number from 1 to 5 denotes different fine-tuned layouts based on input-parallel and output-parallel. For example, `OUT-1` is the fully output-parallel layout which is used by NeuralCache [17] for all layers. The result shows none of layers adopts the original strategy. These results indicate that PIM-DL is applicable to dynamic DPIM DNN inference.

C. Data Layout Aware HW/SW Co-Design

Experiments in Section VI-A3 indicate the software-hardware co-design with optimized data layouts is promising to improve the performance of state-of-the-arts. In this section, we utilize the `Broadcast` interconnect to build customized accelerators based on three widely used DPIM memory technologies: ReRAM, DRAM, and SRAM. We adopt PIM-DL in these customized accelerators and compare the performance with several state-of-the-art accelerators as shown in Table IV. For fair comparisons, we use the same memory size for each proposed architecture (SysR, SysD, SysS) as the state-of-the-

TABLE IV
DPIM SYSTEMS FOR COMPARISON.

Systems	Technology	Mode	Interconnect	Configuration	Area (mm^2)
SysR	ReRAM	Static	Broadcast	32 tiles - 8Gb/tile	40.9
SysD	DRAM	Dynamic	Broadcast	8Gb	31.0
SysS	SRAM	Dynamic	Broadcast	35MB	210.3
FloatPIM [18]	ReRAM	Static	Chain	32 tiles - 8Gb/tile	30.6
Drisa [16]	DRAM	Dynamic	Bus	8Gb	28.5
NeuralCache [17]	SRAM	Dynamic	Bus	35MB	189.8

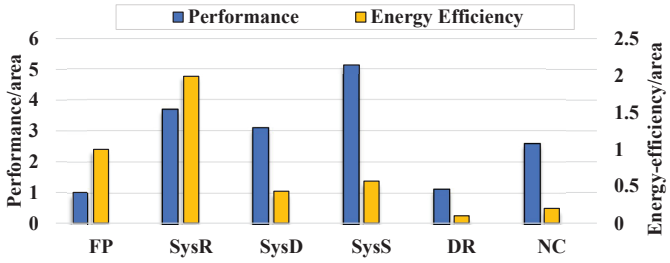


Fig. 17. Performance and energy results of different systems (averaging across all DNNs). All results are normalized to FP and higher values are better.

art using the same memory technology. For static accelerators that cannot fit all layers in the memory (e.g., SysD and SysR), we use a hybrid static/dynamic acceleration and PIM-DL supports this hybrid mode by removing the cross-tile cost in the global optimization. All results are divided by the area of the corresponding systems because of the diversity in system size. All systems run DNN inference workloads with 8-bit fixed-point values. Figure 17 shows the average results across tested DNNs and all values are normalized to SysR.

As compared to the previous accelerator using the same memory technology, each of our customized architectures can provide improvements in both performance and energy efficiency. Specifically, SysR is 3.7× faster and 2.0× more energy-efficient than FloatPIM [18]; SysD is 2.7× faster and 4.3× more energy-efficient than Drisa [16]; SysS is 2.0× faster and 2.8× more energy-efficient than NeuralCache [17]. These results show that the combination of data layout optimization and interconnect customization can significantly improve both performance and energy-efficiency for different technologies and acceleration modes.

We further compare the results across different memory technologies used in our customized architectures. The normalized performance/area improvements of SysR, SysD and SysS are 3.7, 3.1, and 5.14 respectively; the normalized improvements of energy-efficiency of SysR, SysD and SysS are 2.0, 0.4, and 0.6 respectively. Based on such results, the DRAM-based system (SysD) has the worst performance and energy efficiency because of the low density and the large overhead of PIM-enabled circuit [16]. SysS provides the best performance result, which is 1.4× faster than SysR, but it consumes 3.3× more energy/area. Such results indicate Non-volatile memories, like ReRAM, would be more efficient than conventional memory DRAM and SRAM technologies because of its high density and energy efficiency. However, we should note that NVM-based accelerators can only support static acceleration mode because it would be too expensive to frequently load weights through time-and-power-consuming write operations.

VII. Related Work

Memory-centric DNN Accelerators. There are mainly three categories of memory-centric technologies - near-data computing (NDC) [37], [54], APIM [14], [15], [19], [55], and DPIM [16]–[18], which have been extensively explored to accelerate DNN applications. For example, Tetris [37] proposes a scheduling and partition algorithm for a NDC-based DNN accelerator to efficiently map the row-stationary DNN dataflow [41] in 3D stacked memory [46] with maximum data reuse. PUMA [56] is a data-flow accelerator which allocates MVM operations in DNNs on a spatial APIM architecture based on compiler optimizations. The approach proposed by Ji et al. [57] maps NN applications into APIM NN chips. However, the architectures targeted in these work are similar to conventional hierarchical spatial accelerators. As illustrated in Section III, the data layout problem in DPIM cannot be fully represented by such mapping convention.

Data-traffic Optimization for DNN. Because of the large data and model size in modern DNNs, the data traffic has become one of the major bottlenecks in various systems [58]–[62]. HyPar [58] proposes a hierarchical dynamic programming method to determine layer-wise parallelism for deep neural network training with an array of DNN accelerators. The cost models of HyPar are based on partitions of different tensors, which are mapped to the accelerator array. AccPar [63] further supports mapping on heterogeneous accelerators. Tofu [59] automatically partitions DNN models across multiple GPU devices to reduce per-GPU memory footprint as well as the total communication cost. MEDNN [62] optimizes the distribution of DNNs on multiple mobile devices. All these works focus on operation-level partitioning across multiple general-purpose processing units, without further considering data layout. They are orthogonal to the DPIM architectures which require more sophisticated data layout strategies.

VIII. Conclusion

In this work, we comprehensively investigate the data layout issue in DPIM DNN acceleration by representing the layout problem to a mapping framework with both DPIM-specific design dimensions. We then propose an efficient optimization algorithm to generate the good-performing DPIM data layout for general DNN workloads. We conduct several experiments to evaluate the efficiency of proposed data layout optimization and show that PIM-DL provides 3.7× speedup and 4.3× better energy efficiency on a wide range of DPIM DNN accelerators as compared to existing layout strategies.

Acknowledgment

This work was supported by Alibaba Group through Alibaba Research Intern Program. This work was also funded by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, SRC Task No. 2988.001, and NSF grants (#1730158, #2100237, #1911095, #2112167, #2052809, #1826967, #2127780).

REFERENCES

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[2] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *arXiv preprint arXiv:1906.08237*, 2019.

[3] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury *et al.*, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal processing magazine*, vol. 29, 2012.

[4] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 6645–6649.

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

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

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

[8] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[9] A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis," *arXiv preprint arXiv:1809.11096*, 2018.

[10] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 269–284, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2654822.2541967

[11] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 609–622. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.58

[12] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[13] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 336–348.

[14] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 14–26, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/3007787.3001139

[15] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 27–39, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/3007787.3001140

[16] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 288–301. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123977

[17] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 383–396. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00040

[18] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 802–815. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322237

[19] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 541–552.

[20] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 531–543.

[21] L. Yavits, S. Kvatinsky, A. Morad, and R. Ginosar, "Resistive associative processor," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 148–151, 2015.

[22] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *2017 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 2017, pp. 457–468.

[23] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173171

[24] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 316–331.

[25] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, 2018.

[26] M. Zhou, M. Imani, S. Gupta, and T. Rosing, "Gas: A heterogeneous memory architecture for graph processing," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2018, pp. 1–6.

[27] M. Zhou, M. Li, M. Imani, and T. Rosing, "Hygraph: Accelerating graph processing with hybrid memory-centric computing," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 330–335.

[28] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[29] B. C. Jang, Y. Nam, B. J. Koo, J. Choi, S. G. Im, S.-H. K. Park, and S.-Y. Choi, "Memristive logic-in-memory integrated circuits for energy-efficient flexible electronics," *Advanced Functional Materials*, vol. 28, no. 2, p. 1704725, 2018.

[30] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 185–197. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540725

[31] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 100–113. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358260

[32] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 481–492.

[33] M. Zhou, M. Imani, Y. Kim, S. Gupta, and T. Rosing, "Dp-sim: A full-stack simulation infrastructure for digital processing in-memory architectures," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2021, pp. 639–644.

[34] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina *et al.*, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 369–383.

[35] A. Parashar, P. Raina, Y. S. Shao, Y. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.

[36] M. Zhou, M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Gram: graph processing in a reram-based computational memory," in *IEEE Asia and South Pacific Design Automation Conference*, 2019.

[37] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 751–764, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3093337.3037702

[38] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 2010.

[39] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[40] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.

[41] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[42] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.

[43] "DDR3 SDRAM," https://www.micron.com/products/dram/ddr3-sdram.

[44] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.30

[45] A. B. Kahng, B. Lin, and S. Nath, "Explicit modeling of control and data for improved noc router estimation," in *Dac design automation conference 2012*. IEEE, 2012, pp. 392–397.

[46] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *2012 symposium on VLSI technology (VLSIT)*. IEEE, 2012, pp. 87–88.

[47] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[48] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[49] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[50] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[51] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[52] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 105–117.

[53] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 544–557.

[54] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 655–668.

[55] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 1–13.

[56] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 715–731. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304049

[57] Y. Ji, Y. Zhang, W. Chen, and Y. Xie, "Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 448–460. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173205

[58] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Hypar: Towards hybrid parallelism for deep learning accelerator array," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 56–68.

[59] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.

[60] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks," *arXiv preprint arXiv:1802.04924*, 2018.

[61] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *arXiv preprint arXiv:1807.05358*, 2018.

[62] J. Mao, Z. Yang, W. Wen, C. Wu, L. Song, K. W. Nixon, X. Chen, H. Li, and Y. Chen, "Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 751–756.

[63] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Accpar: Tensor partitioning for heterogeneous deep learning accelerator arrays," in *26th IEEE International Symposium on High Performance Computer Architecture, HPCA*, 2020, pp. 22–26.