Memory and Interconnect Optimizations for Peta-Scale Deep Learning Systems

Swagath Venkataramani, Vijayalakshmi Srinivasan, Jungwook Choi, Philip Heidelberger,
Leland Chang, Kailash Gopalakrishnan
IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
swagath.venkataramani@ibm.com, {viji,choij,philiph,lelandc,kailash}@us.ibm.com

***Abstract*—Hardware accelerators are a promising solution to the stringent computational requirements of Deep Neural Networks (DNNs). Ranging from low-power IP cores to server class systems, various accelerator architectures with high TOPS/W peak processing efficiencies and flexibility to execute different DNN topologies have been proposed. Prior efforts improve core utilization through better data-flows and computation sequencing, but little effort has thus far been devoted to systematically programming DNN accelerators to extract best possible system utilization, particularly for DNN training, which can be parallelized across peta-scale systems. In this work, we address the hitherto open challenge of systematically mapping computations onto Peta-scale accelerator systems, comprising many (thousands of) processing cores spanning many chips, while maximizing overall system performance. We achieve this by characterizing the design space of possible mapping configurations, building a detailed performance model that incorporates every computation and data-transfer involved in DNN training, and using a design space exploration tool called DEEPSPATIALMATRIX to identify the performance optimal configuration. We highlight 4 key optimizations built within DEEPSPATIALMATRIX - hybrid data-model parallelism, inter-layer memory reuse, time-step pipelining, and dynamic spatial minibatching - each of which improve system utilization by carefully managing the available memory capacity and interconnect bandwidth to balance the compute *vs.* communication costs. On a 8-peta-FLOP accelerator system, we demonstrate 1.36×-32× improvement in training performance through our design space exploration and optimizations across image recognition (VGG16, ResNet50) and machine translation (GNMT) DNN models.**

***Index Terms*—Hardware Accelerators; Deep Neural Networks;**

I. INTRODUCTION

Advances in Deep Neural Networks (DNNs) have enabled super-human accuracies on several challenging Artificial Intelligence (AI) tasks [1]–[5] and triggered the explosive growth of AI workloads across the spectrum of computing devices. The high accuracy of DNNs come at a high computational cost. For example, state-of-the-art DNNs (*e.g.* ResNet50 [3]) take >10 billion scalar operations to classify a single image. Furthermore, training DNN models require exa-FLOPs of compute and use massive training datasets, which are 100s of GB in size, severely stressing the capabilities of traditional computing platforms. Thus, new approaches are necessary to reduce the training time from weeks/days to hours/minutes.

Hardware accelerators, ranging from low-power cores [6]–[14] to large-scale systems [15]–[17], have emerged as the architecture of choice to address the computational challenges posed by DNNs. The following characteristics make DNNs amenable to specialized systems: (i) computations can be expressed as static data-flow graphs, (ii) computation patterns are regular, predictable, and offer abundant opportunities for data-reuse, and (iii) functionality can be encapsulated within a set of few (tens of) basic functions (*e.g.* convolution, matrix-multiplication, *etc.*). These architectures have demonstrated impressive peak processing efficiencies in the range of 100s-1000s GFLOPs/W leading to over an order of magnitude improvement in training and inference time relative to GPGPU. However, there has been little focus on improving the *system-level utilization* of the architectures to extract the best possible performance, which is the objective of our work.

Programming DNN Accelerators. An important attribute of almost all DNN accelerators is that they are *programmable i.e.,* flexible enough to execute DNNs of various shapes and sizes. In addition to ensuring functionality over a broad range of DNN topologies, programmability is key to extract the best possible performance at the system-level. This is because *while DNNs can be expressed with very few functional primitives, the heterogeneity in the sizes and shapes of the various data-structures constituting each layer renders each operation computationally unique*. For instance, the number of input channels to convolutional layers of VGG16 [2] vary between 3 to 512, while the size of each feature ranges between 224×224 to 14×14. This translates to an order of magnitude difference in the FLOPs/Byte requirement - 842.5 for layer CONV3_2 *vs.* 25.7 for layer CONV1_1 at half-precision - even though each layer performs the same convolution operation. Prior efforts at programming DNN accelerators [8], [18], [19] are limited on three key fronts: (i) they focus on improving core-level utilization through new data-flows and/or careful sequencing of computations, and (ii) they primarily target DNN inference, and (iii) their analysis is restricted to optimizing convolution and/or matrix multiplication for given DNN layer, with little emphasis on inter-layer compatibility and other DNN functions.

In this work, we set out to move beyond a core and *holistically study the performance of training DNN models on large-scale accelerator systems* composed of multiple accelerator chips and thousands of processing cores, together offering peta-FLOPs of peak computing power. We maximize system-level utilization by systematically partitioning and mapping DNNs across the accelerator chips so as to exploit different forms of parallelism available in the application to efficiently balance communication and computation cost. Based on our detailed analysis, we propose new optimizations to better utilize memory capacity and interconnect bandwidth, which further improve system performance. Our contributions are:

- **DeepSpatialMatrix.** We define a design space of possible configurations to map DNNs onto peta-scale DNN accelerators. Specifically, our design space targets the dimensions in which computations can be spatially split across the many (thousands of) compute elements in the architecture. The cost of data-transfer between compute elements and memory is a strong function of how the workload is split. Using an analytical performance model and a design space exploration tool called DEEPSPATIAL-

2640-0316/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPC.2019.00036

MATRIX, we identify the configuration that maximizes overall system performance.

- **Memory/Interconnect Optimizations.** To better utilize the memory and interconnect sub-systems of peta-scale DNN accelerators, we highlight 4 key optimizations for DNN training built within DeepSpatialMatrix *viz. hybrid data-model parallelism, inter-layer memory reuse, time-step pipelining, and dynamic spatial minibatching*. We present execution scenarios under which each technique is most useful.
- **Result Summary.** We study 3 state-of-the-art DNNs for image recognition (VGG16, ResNet50) and machine translation (GNMT). using DEEPSPATIALMATRIX. Our design space exploration and optimizations yield 1.36×-32× reduction in training time on a 8 peta-FLOP accelerator system.

II. DEEPSPATIALMATRIX: DESIGN SPACE CHARACTERIZATION AND EXPLORATION

To assess the performance of training DNNs on large-scale accelerator systems and to identify the performance optimal configuration to spatially map them, we develop a tool called DEEPSPATIALMATRIX. Figure 1 illustrates the key components within DEEPSPATIALMATRIX. It takes descriptions of the workload and the accelerator system as its inputs. The accelerator system is specified in DEEPSPATIALMATRIX by configuring a hierarchical and scalable architectural template (Section II-B), which we believe is general enough to cover a broad range architecture choices. Using these, DEEPSPATIALMATRIX defines a *design space* of possible spatial mapping configurations for each DNN layer and pass (forward or FWD, backward or BWD, and update or UPD). We built an analytical *performance model* within DEEPSPATIALMATRIX that estimates system-level utilization for any given mapping configuration. We accomplish this by identifying the computations performed by each core and the data-transfers engendered between the various system components (some of which could be overlapped with compute) based on how the workload was divided. Finally, the *design space exploration* component runs through the space of mapping configurations and identifies the configuration that yields the best system utilization, which is provided as the output. Starting with how the workload and the architecture are described, this section explains each of the components of DEEPSPATIALMATRIX.

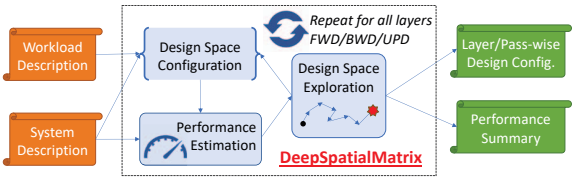

Fig. 1. DeepSpatialMatrix: Overview

A. Workload Description

DEEPSPATIALMATRIX can analyze state-of-the-art DNN models, including convolutional networks for image recognition, recurrent and Long-Short Term Memory (LSTM) networks for natural language and machine translation. We classify the computations in DNNs into two broad categories. (i) *Primary computations viz.* convolution and matrix-multiplication, which occupy >95% of the ops. With high FLOPs/Byte ratios, they offer abundant opportunities for data-reuse and are executed on multi-dimensional computing arrays in DNN accelerators. (ii) *Auxiliary computations*, which are typically performed on the outputs of convolution or matrix multiplication and offer very little data-reuse. Examples include scalar transformations such as activation functions, bias/residue addition, local transformations such as pooling, and normalization operations such as softmax and batchnorm. DNN accelerators typically contain a relatively small number of special function units to execute these computations.

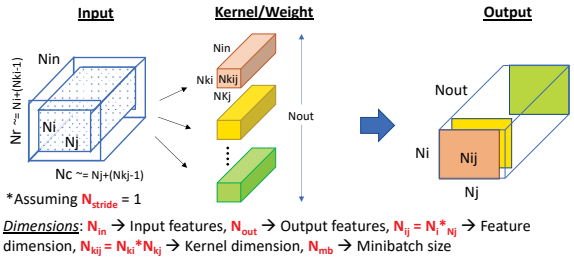

Fig. 2. Describing primary computations of DNNs

Primary Computations. The primary computations use three data-structures: *Input*, *Kernel* or *Weight*, and *Output*, each of which are three-dimensional. As shown in Figure 2, the dimensions can be succinctly expressed with 5 parameters: (i) N_{in}: Number of input features, (ii) N_{out}: Number of output features, (iii) $N_{ij} = N_i * N_j$: Size of each feature, (iv) $N_{kij} = N_{ki} * N_{kj}$: Size of each kernel, and (v) N_{mb}: Size of minibatch. The $Input$ is given by $N_{in} * N_{mb} * N_{ij}$, $Kernel$ by $N_{out} * N_{in} * N_{kij}$, and $Output$ by $N_{out} * N_{mb} * N_{ij}$. While the above parameters provide a compact representation of each layer, a few corner cases exist. For convolutional layers, the zero-padding for the $Input$ can be inferred from the kernel dimensions. A very small fraction of layers employ strided convolution, which we capture using stride parameter N_stride. Fully-connected layers require only a subset of the above parameters, namely, feature size (N_{ij}) and kernel size (N_{kij}) are 1. LSTM layers also have $N_{ij} = N_{kij} = 1$, but they additionally contain a timestep dimension (N_{ts}). Each timestep takes input from the previous layer as well as the previous timestep. Hence the number of input features in LSTMs is effectively $N_{in} + N_{out}$.

Auxiliary Computations. DEEPSPATIALMATRIX enables a set of auxiliary computations to be specified for each layer. These include bias addition, residual connections, activation functions (ReLU, Sigmoid, Tanh), pooling functions (Max and average pooling), and normalization functions (Softmax and batch normalization). Pooling functions include additional parameters to specify size of the pooling window (N_{ws}).

B. Architectural Template

To describe the DNN accelerator architecture, DEEPSPATIALMATRIX uses a flexible, scalable, and highly parameterized architectural template shown in Figure 3. The template comprises of a 5-tiered hierarchy of components. From the ground up, the first level of hierarchy is the *Processing Element (PE)*. Each PE contains a register file and SIMD array of multiply-and-accumulate units. At the second level is *PE Array*, which is formed by a systolic 2D connection of PEs. Convolutions and matrix multiplications are highly

efficient on 2D systolic arrays and are commonly used in many DNN accelerators [8], [13], [17]. The third level is the *Core*, which comprises of multiple PE arrays along with a private scratchpad memory. The core also contains Special Function Processors (SFPs) which carry out auxiliary computations on the outputs produced by the PE arrays.

In the fourth level, multiple cores are connected using an on-chip interconnect network to form a *Chip*. The chip is interfaced with a high bandwidth external memory. At the chip-level, DEEPSPATIALMATRIX supports ring and split-bus interconnects. Finally, multiple chips are connected as a 2D-torus to form the accelerator *System*. The choice of a 2D-torus was borne out of both workload and hardware considerations. As it will be described in Section II-D, the two key functions which involve chip-to-chip communication are gradient reduction and feature rotation, both of which are optimal on a 2D-torus. From a hardware perspective, a 2D-torus is simple, power-efficient and resilient to link failures.

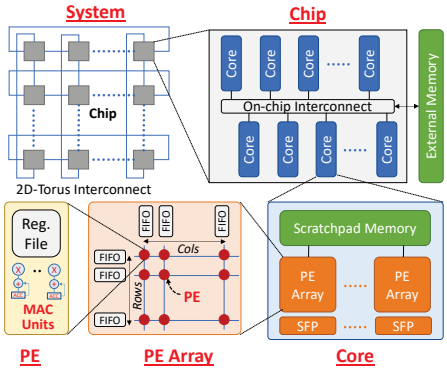

Fig. 3. DEEPSPATIALMATRIX architectural template

Configurable Parameters. To study accelerator designs of varying compute characteristics using DEEPSPATIALMATRIX, the architectural template enables configuration of its compute, memory, and interconnect parameters. **Compute parameters** include number of SIMD lanes in the PE, number of rows and columns in the PE array, number of PE arrays and SFPs in the core, number of cores in the chip, and number of chips (along both X- and Y-directions of the 2D-torus) in the system. **Memory parameters** include size of register file in PE, scratchpad memory capacity in core, external memory capacity in chip. **Interconnect parameters** include, bandwidth of on-core interconnect between PE array/SFP and scratchpad memory, bandwidth of on-chip interconnect network, and bandwidth of off-chip links in the 2D-torus (along X- and Y-directions).

C. Design Space Characterization

Based on workload and architecture descriptions, we identify a design space of mapping configurations, shown in Figure 4, which defines how the computations are spatially divided across the various compute elements in the architecture. For each layer and pass (FWD/BWD/UPD), we start with the $N\{in, out, ij, kij, mb\}$ parameters, which define the workload. The N parameters are divided into $ChipD\{in, out, ij, kij, mb\}$, which defines the work assigned to each chip. $ChipD$ is further broken into $CoreD\{in, out, ij, kij, mb\}$ and subsequently into $ArrayD\{in, out, ij, kij, mb\}$, which define the work allocated to each core and PE array, respectively.

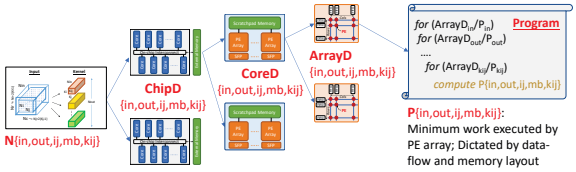

Fig. 4. DEEPSPATIALMATRIX design space parameters

Given the $ArrayD$ parameters, the PE array typically executes the assigned work in smaller chunks across multiple nested iterations (Figure 4). Prior efforts have extensively studied data-flows, memory layouts and methodologies to sequence work that maximizes array-level utilization [8], [18], [19]. Since such optimizations are outside the scope of our work, we encapsulate their effects by defining $P\{in, out, ij, kij, mb\}$, which represents the minimum amount of work needed to maximally utilize the PE array. For consistency, along each dimension, $N \geq ChipD \geq CoreD \geq ArrayD \geq P$. If $N < P$ for any dimension, DEEPSPATIALMATRIX appropriately pads those dimensions.

Parallelism Type (N → ChipD). The division of work across chips is commonly referred to as selection of *Parallelism Type*. The two well known choices of parallelism are data parallelism and model parallelism. In data parallelism, work is divided along the mb dimension, whereas in model parallelism, work is divided along the out dimension. It is non-optimal to split along dimensions unrelated to *Output viz.* kij and in. This is because each chip would then compute only a partial sum of the $Output$, which needs to be reduced across chips, leading to additional data-transfers. Also, it is not preferred to split along the ij dimension, as when the *Output* is used as *Input* to the next layer, border elements of the *Output* need to be communicated between chips to ensure the *Input* covers the full *Kernel* window. Therefore, we limit the ChipD division to only mb (data) and out (model) dimensions. However, as an additional optimization, we explore combinations of data and model parallelism (which we call hybrid parallelism), which is explained in Section III-A.

In summary, the design space comprises of 11 parameters, *viz.* 1 *parallelism type* parameter, which defines the work division across chips ($ChipD$), 5 $CoreD$ parameters and 5 $ArrayD$ parameters.

D. Performance Model

We next describe how DEEPSPATIALMATRIX evaluates the performance of a given design space configuration. One of the key challenges is to model the architecture at the right-level of detail, which engenders a trade-off between model accuracy and simulation time. Functional or cycle-by-cycle simulation is an overkill, as there are no data-dependent execution paths in the workload and/or speculative execution in the architecture. They are also prohibitively expensive for large DNNs. Further, data access patterns in DNNs are both static and regular. This allows latency to be effectively hidden by prefetching data through mechanisms such as double buffering. Drawing inspiration from analogous efforts in the HPC domain [20], [21], we adopt a *bandwidth-centric approach*, where we time data-transfers through each link in the system and match them against the time it takes to process the data fetched.

At a high level, our performance model comprises of 4 key components:

- **Primary compute cycles (PriComp).** Time taken by PE array to execute $ArrayD$ work. Our model also includes *startup time i.e.,* the latency it takes bring the first P data into the PE array, so that its execution can begin.
- **Auxiliary compute cycles (AuxComp).** Time taken by SFPs to execute auxiliary computations. Since these computations require high bandwidth from the local scratchpad, we assume the PE array is idle during this time.
- **Overlapped data-transfer cycles (OvlDt).** Time for data-transfers that can be overlapped with primary compute.
- **Non-overlapped data-transfer cycles (Non-OvlDt).** Time for data-transfers that cannot be overlapped with primary compute.

Each data-transfer may use one or more links, and multiple data-transfers can happen concurrently. In this case, we group simultaneous data-transfers into a *data-transfer group* and compute the amount of data carried by each link for that group. The total time is determined by the link which is the slowest.

The total execution cycles is given by:
$TotCy$ = MAX$\{PriComp, OvlDt\}$ + $AuxComp$ + $Non\text{-}OvlDt$

Type	Links Used	Data Transfer			
		Name	Pass	Function	Compute Overlapped?
Core ⇔ External Mem.	On-chip interconnect, Memory bus	FetchInp	FWD, BWD, UPD	Fetch activations/errors and weights for convolution, matrix multiplication. Write outputs back to memory.	Yes
		FetchKer			
		WriteOut			
Core ⇔ Core	On-chip interconnect	PartialSumAcc	FWD, BWD, UPD	Accumulate across cores: partial sum of (i) activations, errors or gradients for Conv/Mat. mul, (ii) exponent for Softmax, and (iii) mean, variance for BatchNorm	No
Chip ⇔ Chip	Chip-to-Chip links, on-chip interconnect	SumofGradAlongX	UPD	[Only data parallelism] Gradient Reduction and Weight Distribution: Reduce partial weight gradients along X,Y directions and broadcast new weights to all chips	No
		SumofGradAlongY			
		BroadcastWAlongX			
		BroadcastWAlongY			
		RotateAlongX	FWD/ BWD/ UPD	[Only Model parallelism] Rotate Activations/Errors across chips so that each chip computes on the complete set of activations/errors	Yes (FWD, UPD) No (BWD)
		RotateAlongY			
	Chip-to-Chip links, on-chip interconnect, Memory bus	RelayoutAlongX	FWD/ BWD	Activation/Error re-layout: When switching parallelisms, re-layout activations/errors across memories of different chips	No
		RelayoutAlongY			

Fig. 5. Key data transfers in DNN training

Figure 5 illustrates the key data-transfers in DNN training. We group the data-transfers in three key types. First, *Core-to-Memory* transfers bring *Input*, *Kernel* and *Output* data from/to the external memory for primary compute operations. These transfers are overlapped with computations that happen on the PE array using double-buffering. We also assume data shared by multiple cores are fetched only once from the external memory and multicast *via* the on-chip interconnect network. The next class of transfers occur between cores. They typically involve reduction of partial outputs in the context of both primary computations (*e.g.* accumulate partial sums in convolution and matrix-multiplication) and auxiliary computations (*e.g.* compute exponent sum for softmax). The final, and the most interesting, class of data-transfers involve communication between chips. They can be grouped under 3 functions, which are described below.

Gradient reduction and Weight distribution. When data parallelism is employed, during the UPD pass, the weight gradients produced by minibatch samples in each chip need to be accumulated. Once the updated weights are computed, they need to be broadcast to all the chips. The 2D torus is optimal for this operation. We achieve this by first accumulating in the X-direction and then in Y-direction in a pipelined manner. The updated weights are broadcast back first along Y and then along X direction.

Activation/Error Rotation. When model parallelism is employed, features are distributed across chips. Since all input features need to be evaluated to compute the output of a layer, features from each chip need to reach all other chips. We achieve this by breaking the compute into $M * N$ feature iterations for a system with M chips per row and N chips per column. In each iteration, features in a given chip are rotated once in the X-direction to reach its neighbors. Once every M iterations, features are rotated along the Y-direction so that a row of features reach the neighboring row. Feature rotation can be overlapped with the primary computations in the FWD and UPD pass. In BWD pass, where outputs need to be rotated, it cannot be overlapped.

Activation/Error Relayout. When adjacent layers employ different parallelisms, data re-layout across chips is required. For example, when switching from data to model parallelism, the features in a chip are broken into $M * N$ portions, and one distinct portion is sent to every other chip in the system.

Thus the performance model in DEEPSPATIALMATRIX evaluates the time to execute computations and data-transfers to estimate overall system utilization for a given spatial mapping configuration.

E. Design Space Exploration

For each layer and pass, given the design space and the ability to estimate performance, the task boils down to identifying which configuration yields the best utilization by exploring the design space. To achieve this in a reasonable time, we employ various heuristics, some of which are described below. One of the key principles in our exploration is to balance work across all processing elements and loops. For example, at the core-level, to ensure every core does the same amount of work, we enforce a constraint that the ratio of product of $ChipD/CoreD$ along each dimension should equal the number of cores in the chip. At the PE array level, since it executes computations in chunks of P, we explore configurations which are integral multiples of P. If balanced configurations were not possible due to workload dimensions (N), then we explore configurations that yield the least work imbalance. Another principle in our exploration is to minimize core-to-core reduction data-transfers. We achieve this by primarily splitting work in dimensions related to *Output*.

Thus, given the workload and system description, for each layer and pass, DEEPSPATIALMATRIX explores the space of possible spatial mapping configurations to identify the configuration that yields the best system performance.

III. MEMORY AND INTERCONNECT OPTIMIZATIONS

One of the key hallmarks of DEEPSPATIALMATRIX is its ability to identify the performance optimal spatial configuration given workload and architecture constraints. We use such an optimized configuration as our baseline, and build new memory and interconnect optimizations within DEEPSPATIALMATRIX to further enhance utilization. We highlight 4

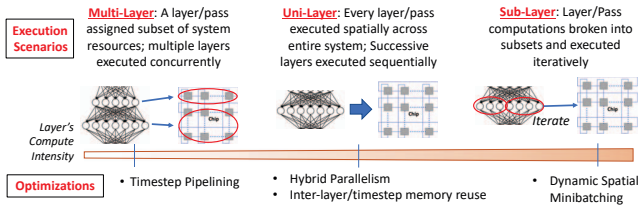

Fig. 6. Overview of optimizations and execution scenarios

key optimizations, which we believe are most useful under 3 different execution scenarios shown in Figure 6.

Uni-Layer Execution. The most common execution scenario is what we call *Uni-Layer*. In DNNs, since each layer depends on the previous layer's output, spatial parallelism is limited to computations within a layer. In Uni-Layer execution, computations of a layer (and pass) are spatially mapped across the entire system and multiple layers are executed temporally in sequence. Uni-layer execution works best when the compute intensity of the layer is commensurate with system's resources, so that each resource is optimally utilized. We study 2 optimizations *viz. hybrid parallelism* and *inter-layer memory reuse* in this context.

Multi-Layer Execution. When the layers are computationally lean, it leads to underuse of system resources, which results in poor utilization. *Multi-Layer* execution addresses this scenario by assigning each layer only a subset of resources in the system and executing multiple layers concurrently. In convolutional models, this is typically not possible due to the producer-consumer relationship between layers. However, in the context of LSTMs and other language models, which have an additional timestep dimension, multiple layers can be executed in parallel provided they are executing different timesteps. We explore an optimization called *timestep pipelining* in this context.

Sub-Layer Execution. At the other end of this spectrum is *Sub-Layer* execution, which is beneficial when the layer is computationally heavy. In this case, some of the system's resources are stressed, which may result in performance loss. For example, the layer may use large data-structures which may spill out of the core's local memory, leading to reduced utilization. To address this scenario, sub-layer execution breaks the layer's computations into subsets, and executes them iteratively. We propose a new optimization called *Dynamic Spatial Minibatching* (DySM) which enables sub-layer execution.

We now describe each of the optimizations in more detail.

A. Asymmetric 2D-Torus and Hybrid Parallelism

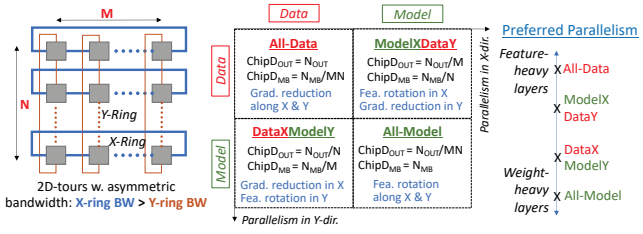

Fig. 7. Hybrid parallelisms in asymmetric 2D-torus

As described in Section II-D, the two prominent chip-to-chip communication patterns are gradient reduction and feature rotation. Depending on the type of parallelism chosen, data *vs.* model, either one of them occurs for each layer. The third pattern, feature relayout, occurs where there is a switch in parallelism between layers, which is quite infrequent. Consider a 2D-torus with M columns and N rows. We note that in both gradient reduction and feature rotation, the amount of traffic along Y-direction is $1/M^{th}$ of the traffic along X-direction. In gradient reduction, since the gradients are first accumulated along X-direction, the number of partial gradient elements shrinks by a factor of M, when the accumulation along Y-direction begins. In feature rotation, rotation along X-direction happens in every iteration, whereas rotation along Y-direction occurs once every M iterations. In light of these observations, we propose to design the 2D-torus *asymmetric in bandwidth*, *i.e., the bandwidth along the Y-direction is* $1/M^{th}$ *of the bandwidth along the X-direction*. For a given total bandwidth coming out of each chip, the re-distribution of bandwidth, more along X-direction compared to Y-direction, would yield performance gains.

Another key advantage of an asymmetric 2D-torus is that it enables two other hybrid forms of parallelism - DataXModelY and ModelXDataY - which are a combination of both data and model parallelism, as illustrated in Figure 7. In DataXModelY, each chip processes $1/N^{th}$ fraction of features for $1/M^{th}$ of mini-batch inputs. In this case, gradient reduction happens *only* along the X-direction and feature rotation happens along the Y-direction. Conversely, for ModelXDataY, each chip is assigned $1/M^{th}$ and $1/N^{th}$ of features and minibatch respectively, which results in feature rotation along X-direction and gradient reduction along Y-direction.

Typically, feature-dominant layers prefer data-parallelism because feature rotation is expensive, while weight-dominant layers prefer model parallelism to avoid gradient reduction. Hybrid forms of parallelism benefit layers that lie in the middle of this spectrum. Small fully connected layers (*e.g.,* last layers of VGG/ResNet), which are moderately weight-heavy, prefer DataXModelY, for they realize gradient reduction along fast X-direction links. On the other hand, convolutional layers with small feature sizes (*e.g.* final set of convolutional layers in ResNet and GoogLeNet), which are marginally feature-heavy, prefer ModelXDataY parallelism to optimize activation/error rotation. Interestingly, LSTM layers, although weight-heavy, prefer ModelXDataY parallelism. In this case, gradient reduction is quite infrequent (occurs only after all timesteps are complete) and can happen on the slow Y-direction links without impacting performance.

B. Inter-layer Memory Reuse

The next optimization focuses on improving utilization by reusing data-structures across layers. Specifically, we build the following memory optimizations within DEEPSPATIAL-MATRIX.

1) Activation/Error Reuse: In DNNs, each layer consumes the output produced by the layer preceding it. A layer's output need not be written back to the external memory and can instead be held in the on-chip memory, provided the core's scratchpad memories have enough capacity, and can then be used by the next layer. Once the next layer consumes the output, they can be discarded (not saved for future) in the case of inference. For training, the output can then be written back to the external memory when the next layer is in progress

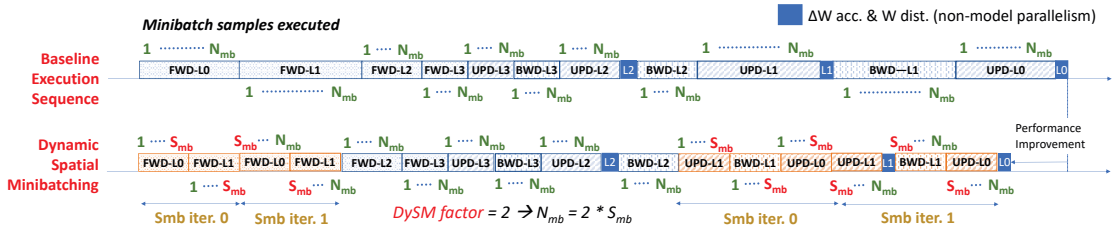

Fig. 8. Dynamic spatial minibatching

so that the UPD pass can use the output. In effect, this optimization saves an external memory read and write for inference and a read for training for the entire data-structure.

While simple in concept, there are several practical considerations. First, the $CoreD$ parameters of a layer determine the format in which its output is produced in the core's scratchpad memory. This influences the work division for the next layer, as it needs to be compatible with the way output is stored in the core's scratchpad memory to avoid additional core-to-core transfers. Second, this optimization is not feasible when adjacent layers use different forms of parallelism, as data relayout in the external memory is required. Also, in the case of residual layers, the outputs may need to be retained longer, as a future layer may also use the outputs for short-cut addition.

BWD/UPD Sequence. In training, the output (error) produced by a BWD pass is used by both the BWD and UPD passes of the previous layer. Typically, there is no preferential order in which these need to executed. However, if feature reuse optimization is enabled, then it is required to execute UPD before BWD pass. This is because if BWD was executed first, then potentially error data-structures for two layers (current and the previous) need to be resident on the core's scratchpad as opposed to just one.

DEEPSPATIALMATRIX takes into account all of the aforementioned practical considerations, including the cost of core-to-core transfers (if any), and aggressively reuses activations and errors across layers to improve utilization.

2) Kernel/Weight Reuse: For convolutional networks, in the context of training, there is no opportunity for reuse of weights beyond the given set of minibatch samples. Hence, we do not hold them on-chip. An interesting opportunity presents in the case of LSTMs and language models, where the same weights can be used across all timesteps. In this case, if the scratchpad memory has enough capacity, when the weights are fetched in the first timestep, we hold them on-chip and reuse them across all timesteps. Since LSTM layers are typically weight-heavy, weight reuse yields a substantial improvement in performance.

Weight Preloading for Inference. The most interesting case of weight reuse occurs in the context of inference. Since end-to-end latency is the key metric for inference, the architecture needs to be optimized to handle very small batch sizes, all the way down to 1. The size of input/output data-structures scale linearly with batchsize (N_{mb}), whereas weights remain the same. Hence, time to fetch weights from external memory becomes the key determinant of utilization. One approach to eliminate this bottleneck is to preload weights on the cores' scratchpads, and use it across several batches of inputs, until a new model needs to be loaded in.

While weight preloading can significantly improve performance, it involves several challenges. First, the entire set of weights may not fit in the core's scratchpad. In this case, a question arises as to which layer's weights should be preloaded to maximize performance. Next, since the scratchpad is private to each core, if multiple cores share weights, then the same value needs to be replicated in each to avoid core-to-core weight transfers. Finally, given a fixed scratchpad capacity, preloading weights may preclude activation reuse for some layers. This leads to an interesting trade-off between which weights and which activations to reuse for maximum utilization. In DEEPSPATIALMATRIX, we sort the layers based on how sensitive their utilization is to fetching weights. Then, we sweep through scenarios where different fractions of weights are preloaded on to the scratchpad (with appropriate replication based on $CoreD$), and identify which configuration yields the best performance.

C. Dynamic Spatial Minibatching

In DNN training, for a given minibatch size, the activation/error data-structures of some layers may not fit within the core's scratchpad memory. This primarily holds true in the context of initial convolutional layers of image models, which are feature-dominant. The problem becomes more severe with recent efforts demonstrating very large minibatch sizes (in the thousands) [22]. We introduce a new technique called Dynamic Spatial Minibatching (DySM) to address this challenge. We illustrate DySM in Figure 8 using a 4-layered DNN as an example. Lets assume the outputs of layers L0 and L1 do not fit in the scratchpad for a given minibatch size N_{mb}. In the baseline execution sequence, each layer (and pass) is executed sequentially for all N_{mb} samples.

In DySM, we break the minibatch into smaller groups called *spatial minibatches* (S_{mb}). The ratio N_{mb}/S_{mb} is called the *DySM factor*. We execute the first S_{mb}, but instead of completing the next S_{mb}, we proceed to the next layer. If we choose S_{mb} to be small enough to fit in the scratchpad, then we can continue to hold them for the next layer, thereby reducing the amount of data-transferred. The choice of S_{mb} cannot be arbitrarily small, as that would reduce the amount of spatial work and the reuse factor for weights, both of which may affect utilization. So it is key to trade-off the competing factors, inter-layer feature reuse *vs.* intra-layer weight reuse, to identify the optimal S_{mb}. Note that DySM does not increase the gradient reduction cost, as gradient reduction is done only after all spatial minibatch iterations are complete (Figure 8).

Thus DySM is an effective approach to benefit from inter-layer memory reuse even when feature sizes are larger than the available scratchpad capacity.

D. Timestep Pipelining

The final optimization, timestep pipelining, is applicable to LSTMs and language models. In LSTMs, the number of neurons per layer per timestep is small, and when parallelized

across a peta-scale system, it leads to poor utilization. In LSTMs, the output of a layer for a given timestep is dependent on the layer's output in the previous timestep and the previous layer's output for the same timestep. This fosters successive layers to be mapped spatially on the system, each simultaneously executing a different timestep in a pipelined fashion. We illustrate this using an example shown in Figure 9.

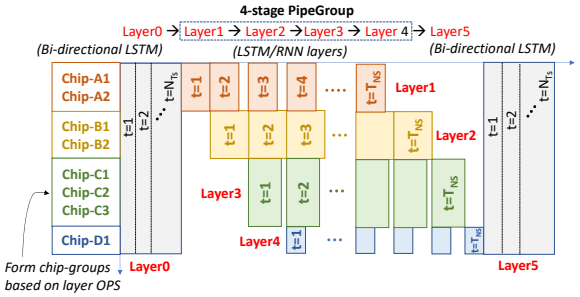

Fig. 9. Timestep pipelining for LSTM and recurrent Nets

Consider a LSTM network with 6 layers. Lets assume Layer0 and Layer5 are Bi-LSTM layers, which is a type of LSTM layer that is not amenable to pipelining. In their case, we map them across all the chips execute each time step sequentially. The middle layers Layer1 to Layer4 are regular LSTM layers, for which we employ timestep pipelining. We call those layers to be part of a pipe group. First, for each layer in the pipe group, we allocate chips based on their OPS complexity. In this case, Layer1 and Layer2 are assigned two chips, Layer3 is assigned 3 chips and so on. The chips assigned for Layer1 execute timestep $t = 1$ and pass their output to chips for Layer2. Then both Layer1 and Layer2 chips begin execution, albeit for different timesteps, $t = 2$ and $t = 1$ respectively. The pipeline builds up and in steady state all the chips are active, each executing a different timestep. We note that timestep pipelining adds a fourth chip-to-chip communication pattern, which is to transfer outputs between adjacent groups of chips.

Thus timestep pipelining enables multiple layers to be mapped spatially onto the system, which enhances utilization.

IV. Experimental Methodology

Architecture Configuration. Figure 10 shows the micro-architectural parameters at system-, chip- and core-levels used in the experiments. We consider a DNN accelerator system of 64 chips, with each chip comprising 32 cores. Our core architecture, based on [13], contains 1024 half-precision floating point (FP16) multiply-and-accumulate engines with 1 MB scratchpad and operates at 2 GHz frequency.

At the chip-level, the cores are connected by a ring and interfaced to a high-bandwidth external memory (HBM) with 8GB capacity and 256 GBps peak bandwidth with an operating efficiency of 80%. Finally at the system-level, the chips are connected as a 2D-torus with 4 and 16 chips along X and Y directions respectively. We assume a bandwidth of 160 GBps coming out of each chip. We study both symmetric (80 GBps along X and Y) and asymmetric bandwidth (120 GBps along X and 40 GBps along Y) configurations. Overall, the system delivers 8 PFLOPs (half-precision) peak processing power. In our experiments, we vary the micro-architectural parameters to study how they impact performance. The ranges used for those sensitivity studies are also shown in Figure 10.

Bold →Baseline configuration; *{italics}* → Range used for sensitivity studies

System Params.	Number of Chips			**64** *{16-256}*
	Chip Params.	Number of Cores		**32**
		Core Params.	Num. of MACs (FP16)	**1024**
			Spad Mem. (MB)	**1** *{0.5-4}*
			Spad. Bandwidth (GBps)	**128**
			Frequency (GHz)	**2**
		Chip Topology		**Ring**
		Core-to-Core Bandwidth (GBps)		**256**
		External Mem. Capacity (GB)		**8**
		External Mem. Bandwidth (GBps)		**256** @ 80% eff.
	System Topology			**2D-Torus** Chips X,Y: **4, 16***{4,64}*
	Chip-to-chip Bandwidth (GBps)			Symm. - X: **80** Y: **80**; Asymm. - X: **120** *{30-240}* Y: **40** *{10-80}*

Fig. 10. Micro-architectural parameters used in evaluation

Benchmarks. Our benchmark suite comprises of 3 state-of-the-art DNNs - 2 convolutional DNNs for image recognition *viz.* VGG16 [2] and ResNet50 [3], and 1 LSTM model called GNMT [5], for machine translation. We selected these benchmarks for their diversity in compute characteristics. VGG16 is 16-layer deep and each convolutional layer is compute-heavy. It also has >100M parameters owing to its large fully-connected layers. ResNet50 is ∼3× deeper than VGG16, but each layer is lean in compute (12× fewer FLOPs) and has <50M parameters. Unlike VGG16, its residual connections introduce data-dependencies between non-successive layers. GNMT has 8 decoder and encoder LSTM layers with the first decoder being a bidirectional LSTM. It operates over 128 timesteps and contains residual connections as well as attention layers. Most layers contain 1024 neurons and are computationally lean with ∼16M FLOPs/layer, which makes them candidates for pipeline parallelism across timesteps.

V. Results

We begin with the results demonstrating the benefits of our design space exploration and optimizations, and follow-up with sensitivity analysis of the overall performance to the different system-level architecture parameters.

A. Training and Inference Performance

1) Training Speedup: Figure 11(a-c) shows the speed-up for training across different minibatch sizes (N_{mb}) normalized to N_{mb} of 256. For each minibatch size, the first bar shows the speed-up achieved using DeepSpatialMatrix's (DSM) design space exploration to distribute work among chips in a symmetric 2D-torus, and the rest of the bars show the improvement in speed-up due to each additional optimizations: asymmetric 2D-torus with hybrid parallelism, inter-layer memory optimizations to enhance feature and weight reuse, dynamic spatial minibatching (VGG16 and ResNet50) or timestep pipelining (GNMT), respectively. Increasing N_{mb}, provides more opportunities to parallelize work, and reduces the frequency of non-overlapped transfers (*e.g.,* gradient reduction) improving utilization. However, as the sizes of datastructures grow with minibatch the pressure on memory capacity/bandwidth increases and negatively impacts utilization.

As the benchmarks are computationally quite heterogeneous, the different optimizations have varying effects on each of them. In the case when DSM is used without any optimizations, increasing N_{mb} yields marginal performance improvement for VGG16. Since layers in VGG16 are computationally heavy, adding more work does not increase utilization. Further, thanks to DSM, its base utilization even for

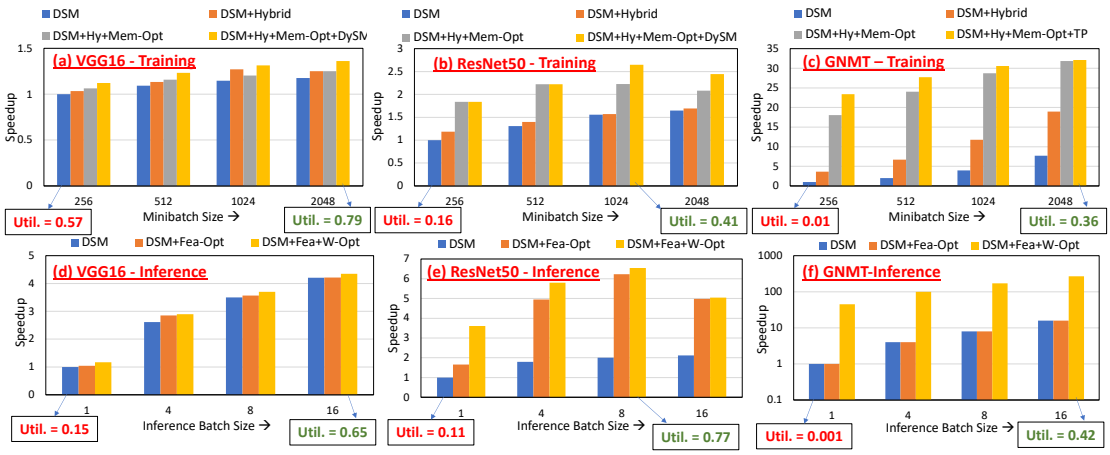

Fig. 11. Speedup achieved through design space exploration and proposed memory/interconnect optimizations

N_{mb} of 256 is reasonably high (0.57). Increasing N_{mb} has a more pronounced performance impact for ResNet50 and GNMT, which are computationally lean.

With an ***asymmetric 2D torus*** and ***hybrid parallelisms*** (Hybrid) enabled, GNMT gains the most (2× - 3×). Its N dimensions are too small to optimally utilize all the processing cores by just splitting work in only one dimension. For a similar reason, ResNet50 under N_{mb} = 256 and 512 also gains from hybrid parallelism (10-18%), which subsides as work per layer is increased. Examining the spatial configurations for ResNet50 revealed that, as per our intuition, the slightly feature-heavy last convolutional layers chose ModelXDataY parallelism to optimize feature rotation over gradient reduction. VGG16's performance gains are marginal (3-12%), which primarily stems from its last two small fully-connected layers (4K and 1K neurons) choosing DataXModelY parallelism.

Memory optimizations (Mem-Opt) are more favorable to ResNet50 and GNMT as they are computationally lean relative to VGG16. For example, the 1x1 convolutions of ResNet50 are memory-bound, and in addition, the features are small enough to fit in the scratchpad, especially for smaller N_{mb}, leading to substantial improvement (55% over hybrid parallelism for N_{mb} = 512). GNMT also benefits from reusing the on-chip weights across timesteps to gain 3× - 4× over hybrid parallelism. ***Dynamic spatial minibatching*** (DySM) further alleviates the memory pressure to better accommodate the data-structures in the scratchpad especially for large N_{mb}. For example, Resnet50 (N_{mb} >= 1024) gains an additional ∼18% speedup relative to hybrid parallelism and memory optimizations combined.

In the context of GNMT, ***timestep pipelining*** (TP) achieves ∼30% additional speed-up (N_{mb} = 256). As the amount of work per layer increases with minibatch size this additional parallelism becomes less effective and the improvement drops to ∼17%. Overall, on a 8 peta-FLOP accelerator system, across all minibatch sizes, we achieve best case utilization of 0.79 for VGG16, 0.41 for ResNet50 and 0.36 for GNMT. The memory and interconnect optimizations boost speedup by 1.36× for VGG16, 2.6× for ResNet50, and 32× for GNMT.

2) Inference Speedup: Figure 11(d-f) shows the speedup for inference across different minibatch sizes. Since inference uses very small batch sizes (1 to 16 in our case), we consider a 128 tera-FLOP system with a single chip in our experiments. In Figure 11(d-f), the first bar represents DSM with no optimizations, and the rest of the bars show the improvement in speedup due feature optimizations (holding features on-chip across layers) and preloading weights in scratchpad before the start of the inference job, respectively.

Feature optimization reduces memory bandwidth pressure by retaining the features on-chip across layers and is therefore more favorable for computationally lean networks. Accordingly, ResNet50 gains 2× - 3× speedup from this optimization. In contrast, VGG16 being compute-dominated and GNMT having small feature sizes do not see significant gains. ***Weight preloading*** eliminates the overhead of fetching weights from memory. This is critical because with very small batch sizes, weight reuse is quite limited. It is worthy to note that it may not be possible to hold the weights of the entire network in the scratchpad. In this case, DEEPSPATIALMATRIX explores the overall weight distribution and determines the set of weights to be pre-loaded to optimize the overall performance. GNMT, which is composed of weight-dominant LSTM layers, benefits the most from weight preloading (∼44× for minibatch = 1). On the other hand, VGG16 does not benefit from weight preloading as well because the large weights of the fully-connected layer do not fit in the scratchpad. ResNet50 exhibits an interesting trend. For small batch size, it gains additional 2× speedup from weight pre-loading. However, as the batch size is increased, the growth in working set size, precludes feature optimizations in the presence of weight preloading (and vice versa), resulting in a net drop in utilization. Overall, even for batch size = 1, the feature and weight optimizations reduce inference latency by 16% for VGG, 3.6× for ResNet50, and 44× for GNMT.

B. Deep-Dive into Spatial Configurations for VGG16

Using VGG16 with a minibatch of 512 as a representative example, we present the spatial configurations chosen by DEEPSPATIALMATRIX for each layer and discuss its impact on utilization for training. Overall, the configurations for FWD/BWD/UPD passes are similar.

Per-Layer Spatial Configurations. Figure 12 shows the performance optimal work split among all the cores in the system. At the system-level, data parallelism is chosen for the compute-dominated convolution layers, and model parallelism is chosen for the largest fully-connected layer (FCON_1),

which is responsible for 90% of the total weights. DEEPSPATIALMATRIX selects the hybrid DataXModelY for the smaller fully-connected layers. The division of work across cores is shown as the $CoreD$ split in Figure 12. At the core-level, work is split along feature dimension (*ij*) for the initial layers and, as the number of feature maps increase in the middle layers, the work is split along feature map dimensions (*in* or *out*). For the fully-connected layers, since work is split along the *in* or *out* dimensions across chips, it is divided along (*mb*) dimension across cores to allow for weight sharing. As most of the convolution layers are not weight dominated, holding features on-chip is more critical than reusing weights. Dynamic spatial minibatching enables features to be held on-chip for all the layers except CONV1_2 which has the largest combined size of the data structures.

Layer FWD/BWD/UPD	Parallelism Type	CoreD Split					Mem-Opt	DySM Factor
		in	*out*	*ij*	*mb*	*kij*		
CONV1_1	Data			32			Y	4
CONV1_2	Data			32			N	4
⋱								
CONV2_2	Data		2	8	2		Y	4
CONV3_1	Data	4	4	2			Y	4
⋱								
CONV4_3	Data		8	2	2		Y	4
CONV5_1	Data	8	4				Y	4
⋱								
FCON1	Model				32		Y	1
FCON2	DataX-ModelY		4		8		Y	1
FCON3	DataX-ModelY				32		Y	1

Fig. 12. Layer-wise design configurations for VGG16

Per-Layer Utilization Impact. Figure 13 shows the utilization of the system for each layer of VGG16 for the FWD/BWD/UPD passes of training. In each stacked bar, we show the percentage of execution time when the MAC units (primary compute) are busy and the additional time due to each of the following overheads: underuse of the PE array due to data-flow, overlapped (double-buffered) data transfer not hidden, non-overlapped data transfer, and auxiliary computations. The overheads and the associated loss in utilization are similar for the FWD and BWD passes, but operations such as gradient reduction show a different distribution of loss in utilization in the UPD pass. In the FWD/BWD passes: (a) Activation function consumes 17% of the total time for *CONV1_1* due to the large feature size, (b) For *CONV1_2* visible overlapped data transfer consumes 26% of the total time because outputs of both the first and the second layer are written back to the external memory, (c) The remaining convolution layers are compute-bound and achieve ∼90% utilization, (d) 22% of execution time is consumed by non-overlapped data relayout for *FCON1* because of a change in parallelism from data to model, and (e) ∼60% to ∼80% of the execution time of the fully-connected layers is due to visible overlapped data transfer for feature and error rotation among the chips. In the UPD pass: (a) gradient reduction is memory-bound especially for later convolution layers (*CONV5_3* to *CONV4_2*) and causes 30% to 60% loss in utilization, (b) *FCON2* and *FCON3* reduce gradient reduction overhead by choosing DataXModelY parallelism, whereas it is completely eliminated in *FCON1* due to model parallelism, (c) ∼30% to ∼70% of the execution time of the fully-connected layers is due to visible overlapped data transfer for feature rotation among the chips. Such detailed analysis of the loss in utilization for each layer from DEEPSPATIALMATRIX is critical to enable the refinement of the interconnection topology, bandwidth, and the associated programming model of such peta-scale architectures.

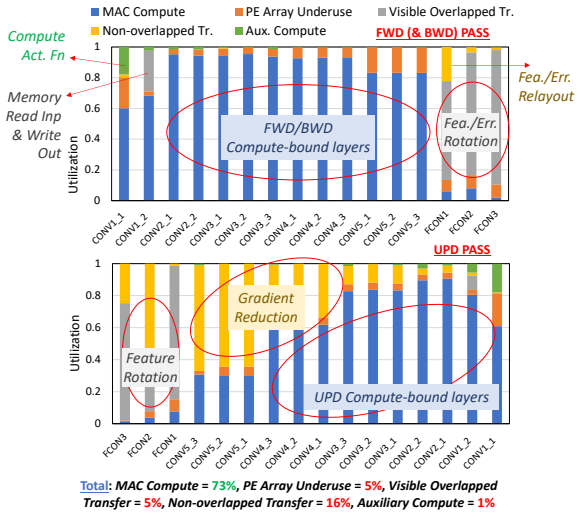

Fig. 13. Layer-wise util. and factors for util. loss in VGG16

C. Sensitivity Studies

In this section, we analyze the sensitivity of performance to architecture parameters by individually varying them.

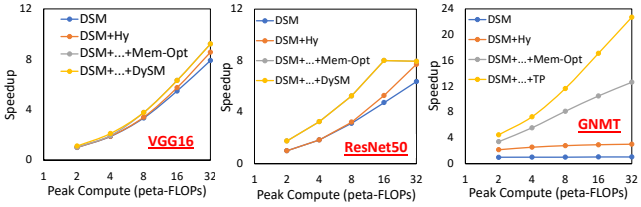

Fig. 14. Performance scaling with peak compute FLOPs

1) Peak Compute FLOPs: Figure 14 shows how performance scales as we vary number of chips (16-256) and peak compute (2-32 peta-FLOPs). While primary computations (CONV and Mat.Mul.) are sped up, performance is limited by other operations such as gradient reduction *etc.*, whose cost may even increase as work is more widely distributed. Thanks to DEEPSPATIALMATRIX's ability to identify the right mapping configurations, we still achieve strong performance scaling. The effect of the optimizations are more prominent at higher peak FLOPs, which increases overall system utilization resulting in super-linear speedups. For example, GNMT achieves 22.7× speedup from 8 to 32 petaFLOP with the optimizations. The compute-heavy layers in VGG16 scale well and the optimizations yield only marginal improvement.

2) Scratchpad Capacity: Figure 15 shows the training utilization as we increase scratchpad capacity from 0.5MB to 4MB. The baseline configuration with no memory optimizations is insensitive to capacity. With ***memory optimizations***, VGG16 with large feature sizes benefits up to 12%. On the other hand, ResNet50 shows a sharp improvement of 37% at 1 MB and saturates thereafter as the feature sizes fit. Interestingly, when DySM is enabled, the upper-bound utilization is reached with much smaller capacity demonstrating DySM's

criticality for capacity constrained architectures. GNMT with very small feature sizes is insensitive to capacity.

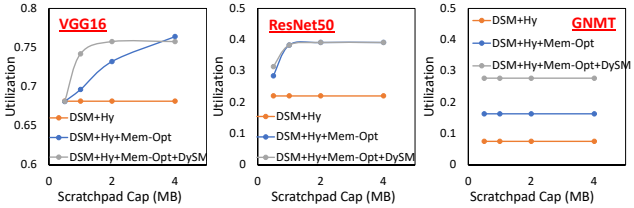

Fig. 15. Sensitivity of training utilization to scratchpad size

VI. Related Work

In this section, we describe prior related research efforts and highlight our distinguishing features.

Programmable DNN Accelerators. A myraid of hardware accelerators, ranging from low-power cores [6]–[13] to large-scale systems [15]–[17], have been proposed to improve the processing efficiency of DNNs. Many of these accelerators employ heterogenous processing tiles and custom ISAs to enable flexible dataflow mapping for programmability. However, they primarily focus on improving core-level or layer-level utilization. In contrast, we studied the performance of training DNNs on peta-scale accelerator systems considering inter-layer optimizations, different forms of parallelism *etc.* to efficiently balance communication and computation cost.

DNN Parallelization and Optimization. DNN workloads have been deployed on peta-scale server systems powered by high-throughput accelerator cores (*e.g.* Google's Cloud TPU Pod [23]), where parallelization and optimization strategies are critical. In the context of CPUs, [24]–[27] provide a good overview of data/model/pipeline parallelization techniques and their tradeoffs. Inter-layer optimizations for additional data reuse opportunities such as feature computation partitioning across layers to retain intermediate output of part of feature-maps for the use as input for the next layer [28], [29] have also been studied. However, a systematic design space exploration combined with a performance model to capture all the computations and data-transfers involved in DNN training including chip-to-chip communications has not been exercised to determine the performance optimal partitioning and mapping of DNNs for such large-scale systems.

RL based Computation Mapping. More recently, DNN computation mapping using reinforcement-learning (RL) has been proposed [30]–[32], where a sequence-to-sequence model trained by policy gradient is used to place DNN computations to GPU devices. However, the complexity of policy gradient is in general significantly increased proportional to the action space [33]. This hinders RL-based methods to scale beyond small numbers of devices. In fact, [30]–[32] take hours for training the RL-model with $<= 8$ GPUs, making them hard to scale for peta-scale systems. In our case, DeepSpatialMatrix using its analytical *performance model* and design space exploration identifies the best mapping configuration within minutes, even for systems comprising of thousands of cores.

VII. Conclusion

DNN accelerators have demonstrated impressive peak processing efficiencies, but very little effort has been devoted towards systematically programming them to extract the best possible utilization. We address this challenge in the context of training DNN models on peta-scale systems. To this end, we propose DeepSpatialMatrix, a systematic methodology that identifies the performance optimal spatial configuration to map each DNN layer and pass. We enhance DeepSpatialMatrix to study 4 different memory and interconnect optimizations, each of which are most effective under different execution scenarios. On a 8 peta-FLOP accelerator system, we demonstrate 1.36×-32× reduction in training time on state-of-the-art DNNs for image recognition (VGG16, ResNet50) and machine translation (GNMT).

References

[1] A. Krizhevsky et. al. Imagenet classification with deep convolutional neural networks. In *Proc. NIPS*, page 2012.
[2] K. Simonyan et. al. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.
[3] K. He et. al. Deep residual learning for image recognition. *arXiv*, 2015.
[4] A. Hannun et. al. Deep speech: Scaling up end-to-end speech recognition. *arXiv*, 2014.
[5] Y. Wu. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv*, 2016.
[6] C. Farabet et. al. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Proc. CVPR workshops*, 2011.
[7] T. Chen et. al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. ASPLOS*, 2014.
[8] Y. Chen et al. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proc. ISCA*, 2016.
[9] S. Han et. al. Eie: Efficient inference engine on compressed deep neural network. In *Proc. ISCA*, 2016.
[10] B. Reagen et. al. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proc. ISCA*, 2016.
[11] J. Albericio et. al. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Proc. ISCA*, 2016.
[12] D. Kim et. al. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *Proc. ISCA*, 2016.
[13] B. Fleischer et. al. A scalable multi?teraops deep learning processor core for ai training and inference. In *Proc. VLSI*, 2018.
[14] S. Venkataramani et. al. Deeptools: Compiler and execution runtime extensions for rapid ai accelerator. *IEEE Micro*, Sep. 2019.
[15] A. Majumdar et. al. A massively parallel, energy efficient programmable accelerator for learning and classification. *ACM TACO*, 2012.
[16] N. Jouppi et. al. In-datacenter performance analysis of a tensor processing unit. In *Proc. ISCA*, 2017.
[17] S. Venkataramani et. al. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *Proc. ISCA*, 2017.
[18] S. Venkataramani et. al. Poster: Design space exploration for performance optimization of deep neural networks on shared memory accelerators. In *Proc. PACT*, 2017.
[19] H. Kwon. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In *Proc. ASPLOS*, 2018.
[20] X. Wu. Performance evaluation, prediction and visualization of parallel systems. *Springer Publications*, 1999.
[21] D. J. Kerbyson et. al. Predictive performance and scalability modeling of a large-scale application. In *Proc. SC*, Nov 2001.
[22] P. Goyal et. al. Accurate, large minibatch SGD: training imagenet in 1 hour. *arXiv*, 2017.
[23] Google cloud tpu: https://cloud.google.com/tpu/, 2018.
[24] T. Ben-Nun et. al. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *arXiv*, 2018.
[25] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv*, 2014.
[26] J. Dean et. al. Large scale distributed deep networks. In *Proc. NIPS*, pages 1232–1240. 2012.
[27] D. Das et. al. Distributed deep learning using synchronous stochastic gradient descent. *arXiv*, 2016.
[28] M. Alwani et. al. Fused-layer cnn accelerators. In *Proc. MICRO*, pages 1–12, 2016.
[29] Lym et. al. Mini-batch serialization: CNN training with inter-layer data reuse. *SysML*, 2019.
[30] A. Mirhoseini et. al. Device placement optimization with reinforcement learning. In *arXiv*, 2017.
[31] A. Mirhoseini et. al. Hierarchical planning for device placement. 2018.
[32] Y. Gao et. al. Spotlight: Optimizing device placement for training deep neural networks. In *Proc. ICML*, 2018.
[33] G. Dulac-Arnold et. al. Reinforcement learning in large discrete action spaces. *arXiv*, 2015.