SCALEDEEP: A Scalable Compute Architecture for Learning and Evaluating Deep Networks

Swagath Venkataramani[§] Ashish Ranjan[§] Subarno Banerjee[‡] Dipankar Das[‡]

Sasikanth Avancha[‡] Ashok Jagannathan[‡] Ajaya Durg[‡] Dheemanth Nagaraj[‡]

Bharat Kaul[‡] Pradeep Dubey[‡] Anand Raghunathan[§]

[§] School of ECE, Purdue University

[‡] Parallel Computing Lab, Intel Corporation
raghunathan@purdue.edu,bharat.kaul@intel.com

ABSTRACT

Deep Neural Networks (DNNs) have demonstrated state-of-the-art performance on a broad range of tasks involving natural language, speech, image, and video processing, and are deployed in many real world applications. However, DNNs impose significant computational challenges owing to the complexity of the networks and the amount of data they process, both of which are projected to grow in the future. To improve the efficiency of DNNs, we propose SCALEDEEP, a dense, scalable server architecture, whose processing, memory and interconnect subsystems are specialized to leverage the compute and communication characteristics of DNNs. While several DNN accelerator designs have been proposed in recent years, the key difference is that SCALEDEEP primarily targets DNN *training*, as opposed to only inference or evaluation. The key architectural features from which SCALEDEEP derives its efficiency are: (i) heterogeneous processing tiles and chips to match the wide diversity in computational characteristics (FLOPs and Bytes/FLOP ratio) that manifest at different levels of granularity in DNNs, (ii) a memory hierarchy and 3-tiered interconnect topology that is suited to the memory access and communication patterns in DNNs, (iii) a low-overhead synchronization mechanism based on hardware data-flow trackers, and (iv) methods to map DNNs to the proposed architecture that minimize data movement and improve core utilization through nested pipelining. We have developed a compiler to allow any DNN topology to be programmed onto SCALEDEEP, and a detailed architectural simulator to estimate performance and energy. The simulator incorporates timing and power models of SCALEDEEP's components based on synthesis to Intel's 14nm technology. We evaluate an embodiment of SCALEDEEP with 7032 processing tiles that operates at 600 MHz and has a peak performance of 680 TFLOPs (single precision) and 1.35 PFLOPs (half-precision) at 1.4KW. Across 11 state-of-the-art DNNs containing 0.65M-14.9M neurons and 6.8M-145.9M weights, including winners from 5 years of the ImageNet competition, SCALEDEEP demonstrates 6×-28× speedup at iso-power over the state-of-the-art performance on GPUs.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**;

KEYWORDS

Deep Neural Networks, Hardware Accelerators, System Architecture

ACM Reference format:
Swagath Venkataramani[§] Ashish Ranjan[§] Subarno Banerjee[‡] Dipankar Das[‡] Sasikanth Avancha[‡] Ashok Jagannathan[‡] Ajaya Durg[‡] Dheemanth Nagaraj[‡] Bharat Kaul[‡] Pradeep Dubey[‡] Anand Raghunathan[§] [§] School of ECE, Purdue University [‡] Parallel Computing Lab, Intel Corporation. 2017. SCALEDEEP: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
https://doi.org/10.1145/3079856.3080244

1 INTRODUCTION

Deep Neural Networks (DNNs), or large-scale layered networks of artificial neurons, have profoundly transformed the field of machine learning and represent the state-of-the-art in a variety of video, image, audio, and text processing tasks [28–30, 34, 36, 45, 47, 48, 51, 53]. Thanks to a confluence of larger data sets to train on, improved algorithms, and growth in computing capabilities, DNNs have made rapid advances in the past few years and greatly outperform other classes of machine learning algorithms and even humans for many recognition tasks. With the proliferation of intelligent devices and the resulting explosion in various forms of digital data, DNNs are widely deployed in many real-world products and services, including Google's image and voice search, Apple's Siri, Facebook's DeepText

Swagath Venkataramani is currently a research staff member at IBM T.J. Watson Research Center, Yorktown Heights, NY (swagath.venkataramani@ibm.com), and Subarno Banerjee is a PhD student at University of Michigan, Ann Arbor, MI (subarno@umich.edu).

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080244

and DeepFace, Microsoft's Skype Translator, and many others [1–3, 7]. Recent efforts to apply DNNs in the fields of autonomous vehicles, education, healthcare, *etc.*, suggest that DNNs will pervade an even wider range of applications in the future [5, 10].

Computational Challenges. Training and evaluating DNNs is highly compute and data intensive. We believe that two scenarios best exemplify the extreme computational challenge imposed by DNNs: (i) Embedded inference, in which DNNs need to be deployed and evaluated on energy-constrained devices such as phones, wearables and IoT edge devices, and (ii) Cloud-based training, in which DNNs are trained in the cloud using massive amounts of data, requiring very high training times. To quantify the computational requirements of DNNs, we consider OverFeat [45], the winner of the localization task in the 2013 ImageNet [43] competition. OverFeat contains ∼820K neurons and ∼145M parameters (weights), and requires ∼3.3 giga operations for evaluating a single 231x231 input image. Similarly, training OverFeat for 1 epoch on the ImageNet dataset [21] with 1.28 million images consumes ∼15 peta operations. Typical training algorithms, such as Stochastic Gradient Descent (SGD), often take 50-100 epochs to converge, making DNN training an exa-scale compute problem. The computational requirements of DNNs continue to rapidly increase, as higher resolution inputs, larger training datasets and networks of higher complexity (more layers, more features, larger feature sizes *etc.*) are utilized. This is illustrated in Figure 1, which shows $> 10\times$ growth in compute requirements for top ImageNet entries from 2012-2015.

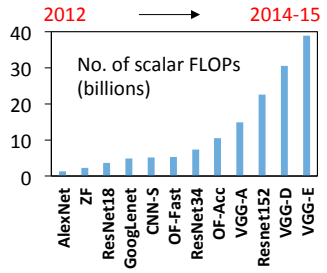


Figure 1: DNN evaluation: FLOPs

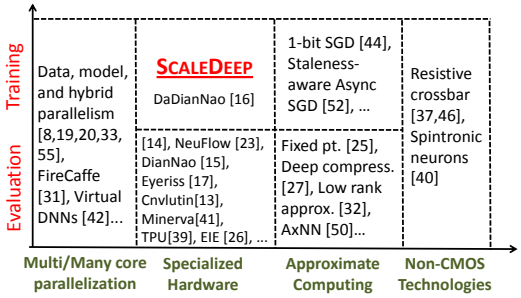


Figure 2: Research directions to improve DNN efficiency

Improving DNN Computational Efficiency. To address the computational challenges imposed by DNNs, prior research efforts have adopted 4 key directions, as summarized in Figure 2. A large fraction of research focuses on parallelizing DNNs on commercial multi-cores and GPUs [8, 19, 20, 31, 33, 42, 55]. However, even state-of-the-art software implementations may take several days to weeks to train large-scale networks, necessitating new approaches for efficiency. To this end, various efforts have explored specialized hardware for DNNs [6, 13–17, 22–24, 26, 38, 39, 41, 49]. SCALEDEEP falls into this category, but differs in two key ways - almost all prior efforts focus on DNN evaluation, and design standalone hardware IPs (cores) that realize key computational kernels within DNNs. In contrast, SCALEDEEP is a full-system (server node) architecture and focuses on the challenge of DNN training. The only effort that addresses training is [15], from which SCALEDEEP differs greatly in its heterogeneity of processing tiles and chips, 3-tiered interconnect topology, and synchronization scheme using data-flow trackers. The third direction is approximate computing, in which the intrinsic ability of DNNs to tolerate imprecise computations is leveraged for efficiency [18, 25, 27, 32, 44, 50, 52, 54]. The final set of efforts explore Post-CMOS devices that intrinsically match the computations present in DNNs [37, 40, 46]. A more detailed comparison of SCALEDEEP with prior efforts is provided in Section 7.

SCALEDEEP **Overview.** We propose a high performance server architecture, called SCALEDEEP, which is designed from the ground-up for DNN training and evaluation. We specialize all subsystems of SCALEDEEP, including compute cores, memory hierarchy and interconnect topology, based on the key computation and data access patterns in DNNs, leading to drastic improvements in performance and energy efficiency. SCALEDEEP differs from prior efforts on DNN hardware in two key ways: (i) SCALEDEEP primarily targets large-scale training of DNNs in contrast to most prior efforts, which focus on network evaluation. Training encompasses network evaluation as one of its steps, but presents a broader set of computational challenges. (ii) While most efforts propose standalone accelerator cores, SCALEDEEP is a scalable server architecture composed of thousands of such processing cores, and deals with system-level challenges such as improving core utilization, memory bandwidth and reducing synchronization overheads. The key features of SCALEDEEP are:

- Heterogeneous processing tiles and compute chips that aggressively exploit the diverse FLOPs and Bytes/FLOP requirements of operations within DNNs.
- A 3-tiered grid-wheel-ring interconnect topology that matches the key memory access and communication patterns of DNNs, including convolution/matrix multiplication followed by feature accumulation within a layer, producer-consumer relationship across layers, and data *vs.* model parallelism across inputs in a minibatch.
- A low-overhead scheme to enforce synchronized execution using hardware data-flow trackers.
- Methods to map DNNs that minimize data movement and improve core utilization through nested pipelining.

We evaluate an embodiment of SCALEDEEP with 7032 processing tiles delivering 680 TFLOPs (single-precision) and 1.35 PFLOPs (half-precision) peak at an efficiency of 485.7 GFLOPs/W. Across a benchmark suite comprised of 11 DNNs with diverse sizes and topologies, including winners from 5 years of the ImageNet challenge (ILSVRC) [43], SCALEDEEP provides $6\times$-$28\times$ improvement in performance over state-of-the-art GPU implementations. While we have extensively benchmarked SCALEDEEP on convolutional neural networks, we note that SCALEDEEP can be programmed to execute other DNN topologies for supervised and unsupervised learning, such as Recurrent Neural Networks (RNNs), Long Short Term Memory (LSTM) networks and autoencoders.

2 DNN COMPUTATIONAL CHARACTERISTICS & CHALLENGES

In this section, we first outline the algorithms used to train and evaluate Deep Neural Networks (DNNs), and identify their key underlying computation patterns. We then present a detailed workload analysis of a state-of-the-art DNN, and use it to quantify computational challenges and highlight opportunities for efficiency.

2.1 Preliminaries

The fundamental compute primitive of DNNs is an *artificial neuron*. Each neuron is associated with a set of parameters or weights. The neuron is a multi-input one-output function, which evaluates a weighted sum of its inputs followed by a non-linear activation function to produce the output. DNNs are comprised of several (millions of) neurons connected to each other. While several variants of DNNs exist in the literature and can be mapped to SCALEDEEP, we focus our explanation on the most popular type, called feed-forward networks, in which the network is organized in layers, with neurons in a given layer connected only to neurons in the successive layer.

There are two key operations involved in DNNs: (i) Training/Learning and (ii) Testing/Evaluation. In the training phase, a training dataset containing inputs labeled with golden outputs is used to learn the network parameters. In the testing phase, the trained network is used to process a new input (*e.g.*, classify into one of several output classes). Training encompasses the steps carried out during network evaluation; therefore, we next focus on describing DNN training in detail.

2.2 DNN Training

Figure 3 hierarchically illustrates the computations involved in DNN training. Given a set of training inputs (F^{L0}) and respective golden output vectors (G^{LN}), the training process learns the weights (W) associated with the connections in the network. Training typically uses a variant of the Stochastic Gradient Descent (SGD) algorithm, which is iterative — the weights are randomly initialized, and in each iteration a training example is used to update the weights of the network such that the difference between the network's output and golden output is minimized. The network-level data flow involved in each iteration of DNN training is shown in Figure 3a. It involves the following 3 key steps:

- ***Forward Propagation (FP)***: Starting with the inputs, neurons in each layer are evaluated in succession until the network outputs are reached. Each layer can start only after the previous layer is complete, but neurons within each layer can be evaluated in parallel.
- ***Backpropagation (BP)***: The error or difference between the golden output and network's output is evaluated. The error is propagated back through the network layers.
- ***Weight Gradient and Update (WG)***: The amount by which each weight in the network is modulated is called the weight gradient. It is computed by accumulating the product of the FP outputs and BP errors corresponding to a given weight. Thus, gradients corresponding to each weight in a layer can be computed in parallel, as soon as the error at the output of the layer is available.

Typically, training iterations over the entire training dataset, called epochs, are repeated several times until the weights converge. Another important concept is that multiple training inputs are grouped into a *minibatch*. The FP/BP/WG steps for the inputs in a minibatch are computed in parallel. After this, their gradients are accumulated together to update the network weights. Algorithmically, as gradients are aggregated, minibatching smoothens convergence. It also presents significant parallelism from a compute perspective. It is worth reiterating that DNN evaluation involves only the FP step.

The computations performed during the FP, BP and WG steps vary based on the *layer type*, which broadly defines how neurons of a layer are connected to the layer inputs. There are 3 key types of layers, and Figure 3b illustrates the layer-level data flow for each layer type.

Convolutional (CONV) Layer. The neurons in CONV layers are arranged as multi-dimensional grids (typically 2D) called *features*. A layer takes multiple input features and produces multiple output features. In the case of FP, each output feature is computed by: (i) convolving each input feature with a weight matrix called the *kernel*, (ii) accumulating the convolved output features, and (iii) performing a non-linear activation function on the accumulated sum. This process is repeated for all output features. In some cases, a connection table denoting which input and output features are connected is specified. The BP/WG steps for CONV layers can be formulated similarly as convolutions followed by accumulations.

Sampling (SAMP) Layer. These layers operate on each feature independently. In FP, output features are produced by down-sampling (max-pooling or averaging) the input features, thus reducing feature size. Similarly, errors are up-sampled during BP. These layers do not contain weights, and hence no WG is performed on them.

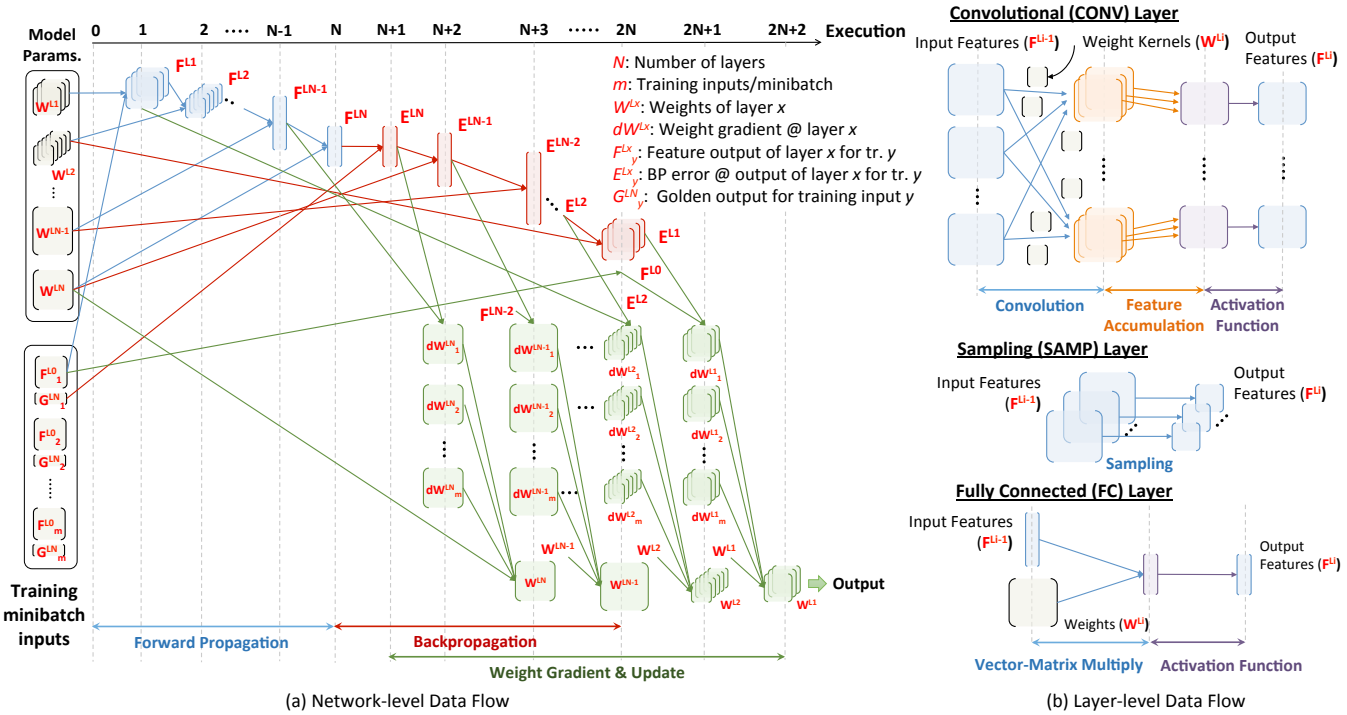
Fully-Connected (FC) Layer. Neurons of a FC layer are organized as a vector. Each neuron is connected to all layer inputs through a distinct weight. The FP/BP steps can be formulated as a vector-matrix multiplication followed by an activation function. The WG step is an element-wise product of the FP output and BP error vectors.

Typical DNNs comprise of a series of CONV layers (5-30 in our benchmarks) and a few FC layers (1-3 in our benchmarks). SAMP layers follow some CONV layers. Intuitively, the CONV layers extract local features by moving the kernel across the input features. As we progress through the DNN, lower-level features (*e.g.*, edges) are composed to construct higher-level features (*e.g.*, complex shapes), which the FC layers use to classify the input.

2.3 Workload Analysis

We quantify the computational characteristics of DNNs in Figure 4, using the *OverFeat* DNN as a representative example. OverFeat comprises of 11 layers - 5 CONV (C1-C5), 3 SAMP (S1-S3) and 3 FC (F1-F3) layers. Like many other DNNs, OverFeat exhibits *significant heterogeneity in computations* both within a layer and across layers.

Inter-layer Heterogeneity. To demonstrate the heterogeneity across layers, Figure 4 lists the compute and data requirements for each layer type. We split the CONV layers into 2 classes, *viz.* initial CONV layers (C1,C2) and mid CONV layers (C3-C5). This is because we find that the initial CONV layers have a relatively small



(a) Network-level Data Flow (b) Layer-level Data Flow

Figure 3: Hierarchical illustration of computations involved in DNN training

number of features (∼4×-11×), each of which is larger (∼4×-16×) in size. In contrast, the mid CONV layers operate on a large number of smaller sized features. As feature count determines the number of convolutions and consequently the number of weights, the mid CONV layers have ∼16×-30× more weights compared to the initial CONV layers. However, its worth noting that each convolution in the initial CONV layers involves 50×-300× more computations, owing to its larger feature size. The FC layers are the smallest in terms of neuron count (1000s), but contain ∼10× more weights than all other layers, as each connection carries a distinct weight. In the case of a SAMP layer, the feature count equals that of the preceding CONV layer, and the feature sizes are smaller by the SAMP window dimensions.

Type	Initial Conv.	Mid Conv.	Fully Conn.	Sub Samp.
Layer	Input, C1, C2	C3, C4, C5	F1, F2, F3	S1, S2, S3
Feature Count	3-256	256-1024	1K-4K	96-1024
Feature Size	24x24 - 231x231	12x12	1	12x12 - 56x56
Weights	30K-600K	1M-10M	4M-120M	---
Data (SP Float)	Feature 560K - 1.14M; Weight 132K - 2.3M	280K – 560K; 4.5M – 36M	12K – 4K; 15M – 432M	140K – 290K; ---
FLOPs FP+BP	11%	54%	3%	0.1%
Bytes/FLOP	0.006	0.015	2	5
WG	5%	26%	0.9%	
	0.005	0.014	4	

Conv. 98.3 Acc. 1.6 Act. 0.1 | Conv. 94.6 Acc. 5.3 Act. 0.1 | Mat Mul. 99.9 Act. 0.1 | Up/Down sampling 100

Conv. 100 | Conv. 100 | Vec. Ele. Mul. 100 | Bytes/FLOP: Low, Medium, High

Figure 4: Breakdown of compute and data requirements for OverFeat DNN

The heterogeneity in layer topologies naturally translates into differing compute and data requirements. Figure 4 tabulates the FLOPs count and Bytes/FLOP (B/F) required by the different layers in the case of FP, BP and WG. For this analysis, we assume single precision floating point representation for both features and weights. The initial CONV layers contribute ∼ 16% of the overall FLOPs, while the mid CONV layers contribute ∼ 80%. The FC layers are quite small - ∼ 4% FLOPs, while SAMP layers contribute only a negligible amount (∼ 0.1%). In terms of B/F, the initial CONV layers offer the most opportunity for weight reuse, as the feature sizes (convolutions) are significantly larger. The input features are most reused in the mid CONV layers, as their feature count is large. Due to these factors, the CONV layers are the lowest in terms of B/F - 0.006 and 0.015 for the initial and mid CONV layers respectively. FC layers offer no weight reuse and their feature sizes are quite small, resulting in a drastically higher B/F of 2. Each feature in SAMP layers is used only once, and they exhibit the highest B/F ratio of 5.

In summary, the CONV layers account for a significant fraction of the FLOPs while offering significant opportunities for data reuse. The FC layers demand high B/F, but contribute only a small fraction to the total FLOPs. Further, as we progress through the network, from the initial and mid CONV layers to the FC layers, we find increasing feature counts, decreasing feature sizes, and a substantial growth in weights. Strikingly, the B/F ratio varies by ∼3 orders of magnitude.

Intra-layer Heterogeneity. We find that computations within a layer also exhibit heterogeneous characteristics. In the initial CONV layers, convolution accounts for ∼ 98% of the FLOPs, while feature accumulation and activation function evaluation contribute the rest.

The mid CONV layers are still dominated by convolution ($\sim 94\%$), but as feature counts grow, feature accumulation carries more FLOPs. In FC layers, almost all FLOPs result from vector-matrix multiplication.

Type	FLOPs (%)	Bytes/FLOP	Layer Type	Step
nD-Convolution	93.1	0.14	CONV	FP,BP,WG
Matrix Multiply	3.02	2	FC	FP,BP
nD-Accumulate	3.02	4.01	CONV, FC	FP,BP,WG
Vector element-wise multiply	0.75	4	FC	WG
Sampling	<0.1	5	SAMP	FP,BP
Activation Fn.	<0.1	8	CONV, FC	FP,BP

Figure 5: Summary of operations in DNN training

Figure 5 summarizes the key computational kernels of DNNs, and quantifies, across our benchmark suite of 11 DNNs, the fraction of FLOPs and B/F for each kernel, and where they are used in DNN training. We find that nD-Convolution occupies 93% of the total FLOPs with 0.14 B/F. Vector-matrix multiplication used in FC layers accounts for 3% FLOPs and has a relatively high B/F ratio of 2. The third key operation is feature accumulation in the FP, BP steps of CONV and FC layers, which accounts for 3% FLOPs, but has a higher B/F ratio of 4. The other operations *viz.* activation function, sampling and vector element-wise multiplication, contribute $< 1\%$, and have the highest B/F.

We next present the SCALEDEEP architecture that addresses the computational traits of DNNs described above.

3 SCALEDEEP ARCHITECTURE

SCALEDEEP is a dense, scalable server architecture built from the ground-up for training deep networks. We specialize SCALEDEEP's compute cores, memory hierarchy and interconnect topology to leverage the key compute and data access patterns of DNNs. Figure 6 shows the hierarchical organization of SCALEDEEP. From the bottom-up, SCALEDEEP is composed of two types of processing tiles, each containing arrays of processing elements and scratchpad memory. At the next level, the processing tiles are connected to form the SCALEDEEP chip. Multiple chips are organized into chip clusters, and at the topmost level, several chip clusters together form the SCALEDEEP node. In this section, we describe the SCALEDEEP architecture and discuss the sources of its efficiency.

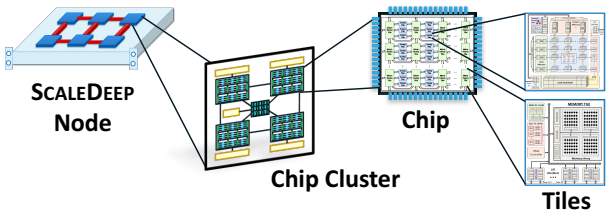


Figure 6: Hierarchical organization of components in SCALEDEEP

3.1 Heterogeneous Processing Tiles

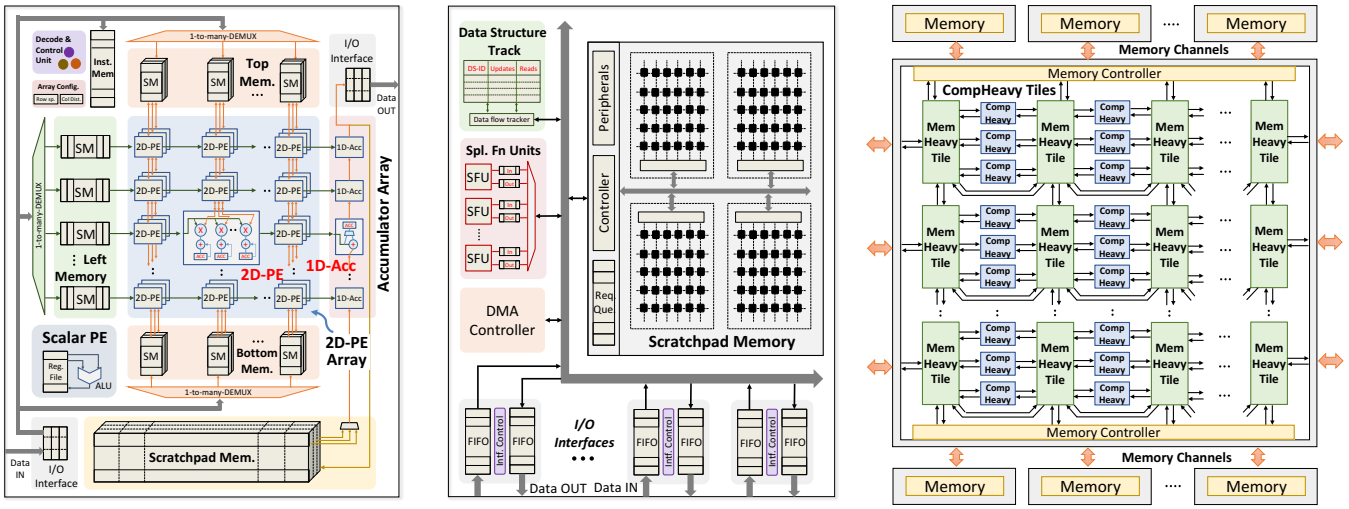
SCALEDEEP aggressively exploits the heterogeneity in the computations within and across layers. We classify the computations in deep networks, shown in Figure 5, into two groups: (i) Operations that are compute dominant and provide ample opportunities for data reuse (low Bytes/FLOP), *viz.* nD-convolutions and matrix multiplications. (ii) Operations that are simple and memory dominant (high Bytes/FLOP ratio) *viz.* nD-accumulation, sub/up sampling, activation function, and vector element-wise multiplication. Accordingly, we design two types of processing tiles - a Compute-Heavy (CompHeavy) tile and a Memory-Heavy (MemHeavy) tile - to realize the compute and memory dominant operations, respectively.

3.1.1 Compute-Heavy Tile. The block diagram of the CompHeavy tile is shown in Figure 7a. It comprises of a reconfigurable 2D array of processing elements (2D-PEs). Each 2D-PE contains a vector of fused multiply and accumulate units. A 1D array of accumulators is located along the right border of the 2D array. Three sets of Streaming Memory (SM) elements are placed along the left, top and bottom borders, using which operands are fed to the 2D-PEs. It also contains a small scratchpad to hold temporary outputs from the 2D array. Other components in the CompHeavy tile include an instruction memory, a decode and control unit, a scalar register file and an in-order scalar processing element to execute control operations such as loop tests, pointer arithmetic, branches, *etc.*

The CompHeavy tile is optimized to carry out batch convolution (one input, many kernels) and matrix multiplication operations. For example, batch 2D-convolution is realized as follows. The rows of the convolution input are fed along the rows of the 2D array and the kernel (weight matrix) rows are fed along the columns. Each 2D-PE computes the dot product of an input row with a kernel row. Convolution output is produced by diagonally accumulating the dot products in the 1D accumulator array. For some convolution outputs, not all row-wise dot products are produced in the same execution iteration of the 2D array. In such cases, the partial outputs are stored in the local scratchpad, and fetched back by the 1D accumulator array when the remaining dot products are available. Since the 2D-PEs have multiple execution lanes, an equivalent number of kernels can be simultaneously fed, enabling multiple convolutions on the same input to be evaluated in parallel.

Array Reconfigurability. The key micro-architectural parameters of the CompHeavy tile are the number of rows, columns and lanes in the 2D array, and the sizes of the left, top, and bottom streaming memories. To improve utilization of the CompHeavy tile, we enable some micro-architectural parameters to be configured at runtime.

- The columns and vector lanes of the 2D array can be redistributed, while maintaining their product to be a constant, *i.e.*, we could dynamically increase (or decrease) the number of columns in the array by proportionately reducing (or increasing) the number of lanes per 2D-PE. This configurability is useful as feature counts and kernel sizes vary across CONV layers, and FC layers involve just a single matrix multiplication (lanes/2D-PE is set to 1).
- The 2D-array can be split horizontally into two 2D-arrays, each with half the number of rows. The SMs on the left are also split in half, while the top and bottom SMs exclusively feed the columns of the smaller 2D arrays. This allows for 2 batch convolutions or matrix multiplications to be executed in parallel. This configurability is desirable to improve core utilization when the CONV feature sizes are not a multiple of the number of rows in the 2D array.



(a) CompHeavy Tile (b) MemHeavy Tile (c) ScaleDeep Chip

Figure 7: Processing tiles and chip architecture in ScaleDeep

3.1.2 Memory-Heavy Tile. The MemHeavy tile, shown in Figure 7b, contains a large scratchpad memory that stores features, weights, errors, and error gradients, and an array of Special Function Units (SFUs) directly interfaced to it. The SFUs contain an adder/comparator, a multiplier, and logic needed to support activation functions such as ReLU, tanh and sigmoid. Operations such as nD-accumulation, activation function, sub/up sampling and vector element-wise multiplication are performed in the MemHeavy tile. The MemHeavy tile includes a DMA controller to enable data transfer to and from the memory array. In addition, the MemHeavy tile includes hardware data flow trackers to track read/write accesses to selected address ranges. Since ScaleDeep does not contain hardware-managed caches or coherence mechanisms, we utilize these trackers (described in Section 3.2.4) to synchronize between processing tiles.

Inst. Type	Instruction	Function
Scalar Control Instructions	LDRI R_d, value	Load scalar register
	ADDR R_d, R_{s1}, R_{s2}	Add scalar registers
	BNEZ R_s, offset	Branch if not zero
Coarse-grained Data Insts.	NDCONV R_{iaddr}, R_{iport}, R_{isize}, R_{kaddr}, R_{ksize}, R_{stride}, R_{pad}, R_{oaddr}, R_{oport}, *isACCUM*	Convolve input with kernel. *isACCUM* → should output be accumulated when stored?
	MATMUL R_{i1addr}, R_{i1port}, R_{i1size}, R_{i2addr}, R_{i2port}, R_{i2size}, R_{oaddr}, R_{oport}, *isACCUM*	Matrix multiplication
MemHeavy tile Offload Insts.	NDACTFN *type*, R_{iaddr}, R_{iport}, R_{isize}, R_{oaddr}, R_{oport}	Compute activation function
	NDSUBSAMP *type*, R_{iaddr}, R_{iport}, R_{isize}, R_{wsize}, $R_{wstride}$, R_{oaddr}, R_{oport}	Subsampling operation
MemHeavy type data-transfer Insts.	DMALOAD R_{saddr}, R_{sport}, R_{daddr}, R_{dport}, R_{size}, *isACCUM*	Load data into MemHeavy tile from another Memheavy tile / external memory
	DMASTORE R_{saddr}, R_{sport}, R_{daddr}, R_{dport}, R_{size}, *isACCUM*	Store data from MemHeavy tile to another Memheavy tile / external memory
Data-flow track Insts.	MEMTRACK $R_{addRange}$, $R_{numUpates}$, $R_{numReads}$	Track accesses to an address range

Figure 8: ScaleDeep ISA

3.2 Chip Architecture

The ScaleDeep chip architecture combines the CompHeavy and MemHeavy tiles to support the diverse computation patterns present in DNNs.

3.2.1 Architecture Description. The architecture of the ScaleDeep chip is shown in Figure 7c. The processing tiles are arranged as a 2D grid, with alternating columns of CompHeavy tiles and MemHeavy tiles. The architecture contains 3 CompHeavy tiles per MemHeavy tile, one each for FP, BP and WG. Each CompHeavy tile is connected to the MemHeavy tiles on its left and right. In addition, each MemHeavy tile is connected directly to its nearest MemHeavy tile neighbors—above and below in the same column, and left and right in adjacent MemHeavy tile columns. External memories are connected to the top and bottom borders of the ScaleDeep chip. We note that all interconnects in the ScaleDeep chip are point-to-point links with no arbitration.

3.2.2 Execution Model and ScaleDeep ISA. In ScaleDeep, we adopt the following execution model. Each CompHeavy tile runs a single thread of execution, whose program is stored in its instruction memory. The memory hierarchy in ScaleDeep is completely software managed, with no traditional caches, coherence, or synchronization mechanisms. The ISA contains 28 instructions, a subset of which is listed in Figure 8. The instructions can be grouped into 5 types:

- **Scalar Control Instructions:** Load/store, scalar ops and branch instructions used to determine program control flow. They are executed on the CompHeavy tile's scalar PE.
- **Coarse-grained Data Instructions:** Compute dominant instructions such as convolutions (NDCONV) *etc.,* executed on the PE arrays of the CompHeavy tile.
- **MemHeavy Tile Offload Instructions:** High Bytes/FLOP instructions such as down-sampling (NDSUBSAMP) *etc.,* offloaded to one of the connected MemHeavy tiles.
- **MemHeavy Tile Data Transfer Instructions:** Instructions that initiate data transfer between connected MemHeavy tiles.
- **Data-flow Track Instructions:** Instructions used to track accesses to specified data structures or address ranges. They are used to enforce synchronized execution (Section 3.2.4).

Since ScaleDeep specifically targets deep networks, whose computations and data flow are static, ScaleDeep can be programmed

with minimal programmer burden. To this end, we develop a compiler front-end to automatically map any DNN topology to SCALEDEEP, which we detail in Section 4.

3.2.3 Implementing DNN Layers on the SCALEDEEP *chip.* We now describe how DNNs are realized on the SCALEDEEP chip. Layers of the DNN are spatially spread across the entire SCALEDEEP chip by allocating a set of columns to each layer based on its compute and memory requirements.

Distributed Network State and Localized Computations. A key aspect of the mapping process is that, at compile time, the entire network state (features, errors, weights, and gradients) is partitioned and distributed across the MemHeavy tiles in the chip. Each feature and error array in the network is assigned a *home* MemHeavy tile. We provision enough memory capacity, cumulatively across all MemHeavy tiles, to hold all the features and errors of the network. State-of-the-art DNNs contain few million neurons, hence their outputs and errors can usually be stored on-chip. In the case when the features and errors cannot fit on a single chip, we split the network and utilize multiple SCALEDEEP chips. We explain this while discussing the node architecture in Section 3.3.

Depending on the memory capacity available, weights and weight gradients for selected layers are stored on-chip, in the MemHeavy tiles where corresponding features reside. Weights and gradients for the other layers are stored in the external memory and prefetched into the MemHeavy tiles. CompHeavy tiles produce and consume the network state stored in MemHeavy tiles directly connected to them. The SFUs present within the MemHeavy tiles also operate on the network state stored in them. Thus, by partitioning the network state and associated computations spatially across the many processing tiles of the SCALEDEEP chip, we localize data movement and minimize interconnect bandwidth. This, coupled with a simplified memory hierarchy, significantly contributes to the energy efficiency of SCALEDEEP.

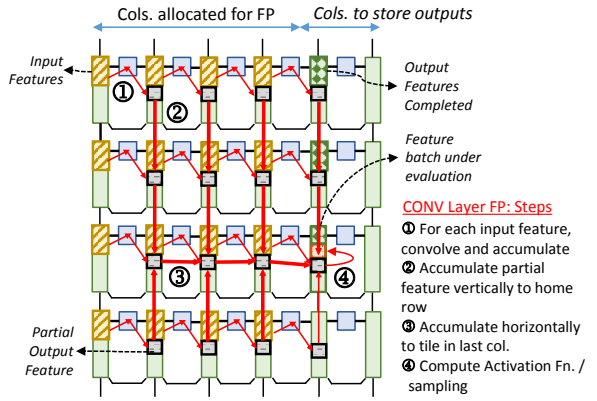


Figure 9: Illustration: Realizing CONV layer FP on a SCALEDEEP **Chip**

Illustration: CONV Layer FP. We now describe how computations of a given layer are realized on a set of chip columns allocated to it, by using the FP step of a CONV layer as an example (Figure 9). First, input features to the layer are distributed evenly across the MemHeavy tiles. Similarly, the output features produced are assigned to the next set of columns, so that the following layer can be computed. The output features are computed in batches of size equal to the number of lanes in the 2D-PEs of the CompHeavy tile. Each output feature batch is computed in four steps. First, each CompHeavy tile fetches an input feature from its left MemHeavy tile, and convolves the input feature to produce partial output features that are stored in the right MemHeavy tile (step ① in Figure 9). This step is repeated for all input features in the left MemHeavy tile and the right MemHeavy tile accumulates the partial output features as they are stored. Next, to produce the final weighted sum, the accumulated partial outputs in each right MemHeavy tile need to be accumulated together. This is achieved by identifying the "home" row of the output feature batch, and first accumulating the features vertically into the home row (step ② in Figure 9) and then horizontally into the last column (step ③ in Figure 9). Finally, the MemHeavy tile in the home row and last column computes the activation function (and performs sampling if necessary) before passing each output feature to its home MemHeavy tile. This process is repeated until all output features are computed. If the layer weights need to be brought in from external memory, the CompHeavy tile issues prefetch requests at the start of the previous output feature batch iteration.

Realizing Layer Sequences. Successive layers of the DNN are mapped to adjacent sets of columns in the SCALEDEEP chip. This leverages the producer-consumer relationship between layers of a DNN. In the case of FP, inputs to the DNN are fetched from external memory by the columns that realize the first layer. From then, the outputs of a layer produced by a set of columns are consumed by the next, until the FP outputs are obtained. The final set of FP tiles also compute the error at network outputs by finding the difference between the golden and FP outputs. BP and WG are realized in a similar fashion, except they use their respective sets of processing tiles and the direction of data flow is reversed. The weight gradients are either stored to the external memory or on-chip, and the errors are discarded after BP/WG of the layer is complete.

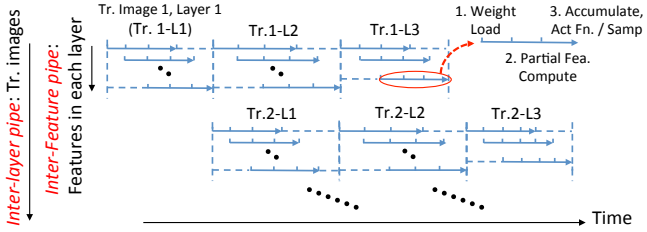


Figure 10: Nested pipelining in SCALEDEEP

Nested Pipelining. To improve utilization of the processing tiles and hide memory latency, the SCALEDEEP chip is programmed (through software executing on each CompHeavy tile) to operate as a 2-level nested pipeline as shown in Figure 10. At the innermost level is the *inter-feature pipeline*, in which computations to evaluate an output feature batch (*viz.* optional weight load, partial feature output evaluation, and feature accumulation/activation function/sampling) form a pipeline. At the outermost level, is the *inter-layer pipeline* in which minibatch parallelism is exploited to operate tiles corresponding to each layer on successive training/evaluation inputs in a pipelined manner. In this case, the pipeline depth is equal to twice the number of layers in the case of training (FP + BP/WG), and equal to the number of layers for evaluation (FP alone). We note that, for training, the

inter-layer pipeline requires the FP features of all layers to be stored in the external memory and fetched back during the corresponding WG step.

3.2.4 Synchronization using Data Flow Trackers. To achieve correct functionality, the programs executing on the different CompHeavy tiles need to be synchronized. Implementing a full-fledged coherence protocol and/or a lock-based synchronization scheme imposes significant energy overheads and is an overkill in our context. We leverage two key insights: (i) the data access sequence to each location in memory can be ascertained at compile time as both the data flow in deep networks and the schemes adopted in the SCALEDEEP compiler to partition computations are static, and (ii) accumulation is commutative *i.e.,* when updates from multiple sources are accumulated at a given memory location, the end result is not affected by the order in which the updates are received. Based on these observations, we propose a new synchronization primitive,

$$MEMTRACK(AddRange, NumUpdates, NumReads) \quad (1)$$

using which software specifies an address range (*AddRange*), number of updates the address range should receive before it can be read (*NumUpdates*), and number of reads before it can be overwritten (*NumReads*). The CompHeavy tile offloads the MEMTRACK instruction to the appropriate MemHeavy tile, which then utilizes hardware counters to track accesses to the address range, and ensures that the access sequence conforms with the specifications. RD (WR) requests that arrive prior to the desired *NumUpdates* (*NumReads*) are queued in the MemHeavy tile, or NACKed on a full queue. In summary, SCALEDEEP enforces synchronization by ensuring that the sequence of accesses to a memory location follows a specified pattern.

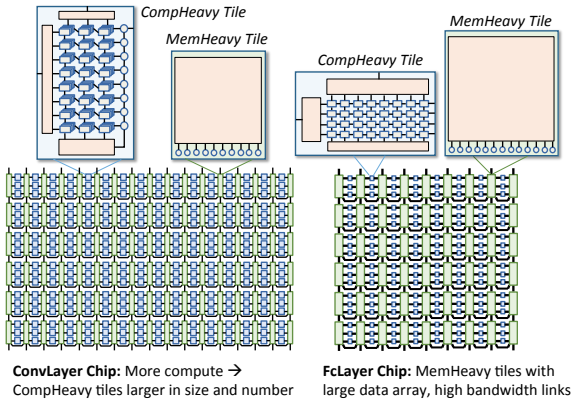


Figure 11: ConvLayer and FcLayer Chips

3.2.5 Heterogeneous Chip Designs. In SCALEDEEP, we use the architectural template described above to build two different chips *viz.* ConvLayer chip and FcLayer chip, shown in Figure 11. To address the diversity between the CONV *vs.* FC layers, the micro-architectural parameters of the chips are tuned differently as follows:

- The FcLayer chip has fewer compute FLOPs. The CompHeavy tiles in this chip have fewer 2D-PEs. To cater to matrix multiplication as opposed to batch convolutions, the 2D array has fewer rows, more columns, and each 2D-PE has only a single lane. Further, the chip contains fewer columns as DNNs in our benchmark suite have relatively few FC layers.
- The memories are organized differently. In the FcLayer layer chip, the scratchpad memory in the CompHeavy tiles is smaller, as FC layers have relatively small neuron counts. On the other hand, the scratchpad in the MemHeavy tiles is larger, but there are fewer of them owing to fewer chip columns.
- The on-chip and off-chip links in the FcLayer chip are designed to support higher bandwidth, as FC layers involve a substantially larger number of weights.

We quantify the micro-architectural parameters of the chips and their on/off-chip bandwidth requirements in Section 5.

3.3 Node Architecture

At the node level, multiple ConvLayer and FcLayer chips are connected in a two-tiered hierarchy, as described below.

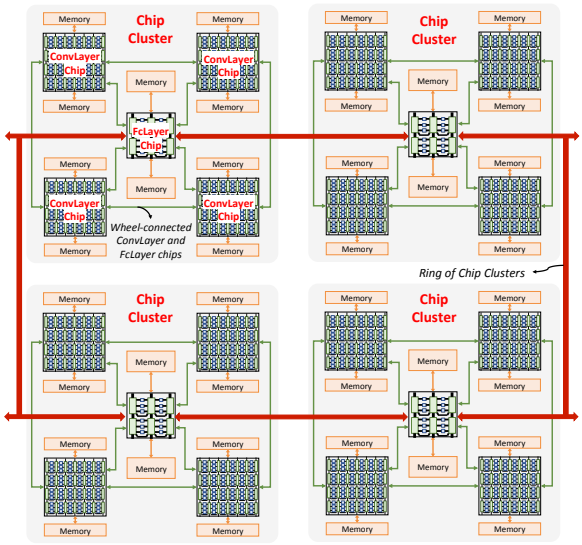


Figure 12: SCALEDEEP: Node architecture

3.3.1 Chip Cluster - Wheel of ConvLayer and FcLayer Chips. One of the key challenges at the node-level is the much higher memory bandwidth required by the FcLayer chips to maintain the same throughput as the ConvLayer chips. One approach to reduce the memory bandwidth is to aggregate inputs to the FC layers, and execute them as a batch in the FcLayer chip. This allows for the layer parameters to be fetched only once per batch, reducing bandwidth proportional to the batch size. Therefore, as shown in Figure 12, we form *chip clusters* by connecting multiple ConvLayer chips and one FcLayer chip as a wheel. The ConvLayer chips are located at the circumference and the FcLayer chip is present at the center. The ConvLayer chips operate on different network inputs in parallel, while the FcLayer chip receives FC layer inputs from all the ConvLayer chips and processes them in a batch. We balance the compute capacity of the FcLayer chip such that it matches the throughput of all ConvLayer chips connected to it. This limits the number of ConvLayer chips in the wheel.

It is worth noting that the wheel configuration reduces the external memory capacity, as the weights of the FC layer (which dominate

the overall weight storage requirement) need to be stored only in the memory connected to the FcLayer chip. Also, note that the arcs of the wheel connect each ConvLayer chip to the ones adjacent to it. This enables the CONV layers to be partitioned across multiple chips in the case they cannot be fit onto a single chip. However, doing so reduces the batch size to the FcLayer chip as only some of the wheel spokes are utilized. The wheel arcs are also used to accumulate weight gradients and distribute updated weights after each minibatch.

3.3.2 Ring of Chip Clusters. To harness further parallelism, we connect multiple chip clusters, through their FcLayer chips, in a ring as shown in Figure 12. Each chip cluster operates on a different set of training/evaluation inputs. In the case of training, at the end of every minibatch iteration, the ring is used to accumulate weight gradients generated at each chip cluster and distribute the updated weights.

The most significant advantage of the wheel-ring configuration is that it enables the FC layers to be parallelized using model parallelism *i.e.*, the parameters of the fully connected layers are split across chip clusters. The features and errors of the FC layers are passed along the ring and evaluated at the appropriate chip cluster. Model parallelism eliminates the need for FC layer weights to be sent along the ring, which greatly reduces the ring bandwidth and also renders the traffic more uniform. Further, model parallelism increases the number of inputs the FcLayer chips can process as a batch, as parts of features from all chip clusters are collected in each FcLayer chip, as opposed to a single chip cluster. Thus, model parallelism reduces the data transferred through the ring, and also lessens the bandwidth of each FcLayer chip.

In summary, the key architectural features of SCALEDEEP are: (i) It leverages the heterogeneity in computations by using two different processing tiles and tuning a common architectural template to realize two different compute chips, (ii) It incorporates a three-tiered grid-wheel-ring interconnect topology to match the communication patterns in DNNs, (iii) It distributes the network state and localizes compute to minimize data movement, (iv) It uses a simplified software-managed memory hierarchy and point-to-point on-chip links to reduce memory, interconnect energies and protocol overheads, and (v) It exploits the static nature of data flow and commutativity of accumulations to alleviate the overheads of synchronization using data flow trackers. By specializing the compute, memory and interconnect subsystems, SCALEDEEP achieves significant efficiency in training and evaluating DNNs.

4 PROGRAMMING SCALEDEEP

To program SCALEDEEP, we develop a compiler front-end that takes a DNN topology and a SCALEDEEP micro-architectural configuration as its inputs, and generates code using the SCALEDEEP ISA for each CompHeavy tile in the design. As shown in Figure 13, the SCALEDEEP compiler works in 2 phases: (i) workload mapping and (ii) code generation, which are described in this section.

4.1 Workload Mapping

The workload mapping phase allocates chip columns to each layer in the DNN, and determines how the network state and computations are distributed across the MemHeavy and CompHeavy tiles of the allocated columns. Figure 13 outlines the steps involved in the workload mapping phase. First, given a DNN topology, we separate the CONV/SAMP and FC layers, and designate them to be realized on the ConvLayer and FcLayer chips respectively (STEP1). We also compute the number of FLOPs required by each layer (STEP2). The next task is to allocate chip columns to each layer (STEP3). To this end, we first compute the minimum number of columns required by each layer, purely based on memory constraints (STEP3a). As the execution is pipelined, the MemHeavy tiles of a layer should cumulatively hold two copies of features and errors, two copies of the partial feature/error batch under evaluation, and corresponding weights and weight gradients. Based on the minimum column constraint we determine the number of chips/chip clusters required to spatially map the DNN. After this, we distribute the extra columns in the chips/chip clusters based on the compute requirements of the layers. To this end, we compute the load per column for each layer as the ratio of the normalized FLOPs to the normalized number of columns allocated to the layer. Additional columns are allocated to the layer with the highest column load (STEP3b).

With the columns allocated, the network state is distributed evenly across all the MemHeavy tiles corresponding to each layer (STEP4). In the case of CONV layers, based on the feature size, the MemHeavy tiles may hold part of a feature or multiple features. Following this, computations are assigned to the FP/BP/WG sets of CompHeavy tiles, based on the features stored in the connected MemHeavy tiles (STEP5). At this stage, we also identify the array configuration for the CompHeavy tiles that yields the best utilization. Finally, based on the memory capacity available for each layer, the weights and weight gradients are deemed to be stored either on-chip or in the external memory (STEP6).

4.2 Code Generation

The code generation phase produces SCALEDEEP programs for each CompHeavy tile in the design. The compiler utilizes a library of hand-coded assembly routine templates for the FP/BP/WG steps of each layer type. These parameterized assembly templates are customized by the compiler based on the information (*e.g.,* features per MemHeavy tile) available from the workload mapping phase. As an example, a code snippet compiled to the SCALEDEEP ISA that computes the FP step of a CONV layer is shown in Figure 13.

5 EXPERIMENTAL METHODOLOGY

In this section, we describe the methodology used in the experiments to evaluate SCALEDEEP.

Performance Evaluation. We develop a detailed and cycle-accurate SCALEDEEP architectural simulator in C++. The simulator rigorously models all events that occur in each execution cycle, including compute operations, on/off-chip memory accesses and data transfers.

Power and Energy Estimation. To estimate compute power, we implemented the CompHeavy and MemHeavy tile execution arrays in RTL, synthesized them to Intel 14nm technology node and measured power at the gate-level using Synopsys Design Compiler [11]. On/Off-chip memory and interconnect power were estimated using in-house models developed for the 14nm node. The power consumed by each component was incorporated into the simulator, and energy is estimated using execution traces observed during simulation.

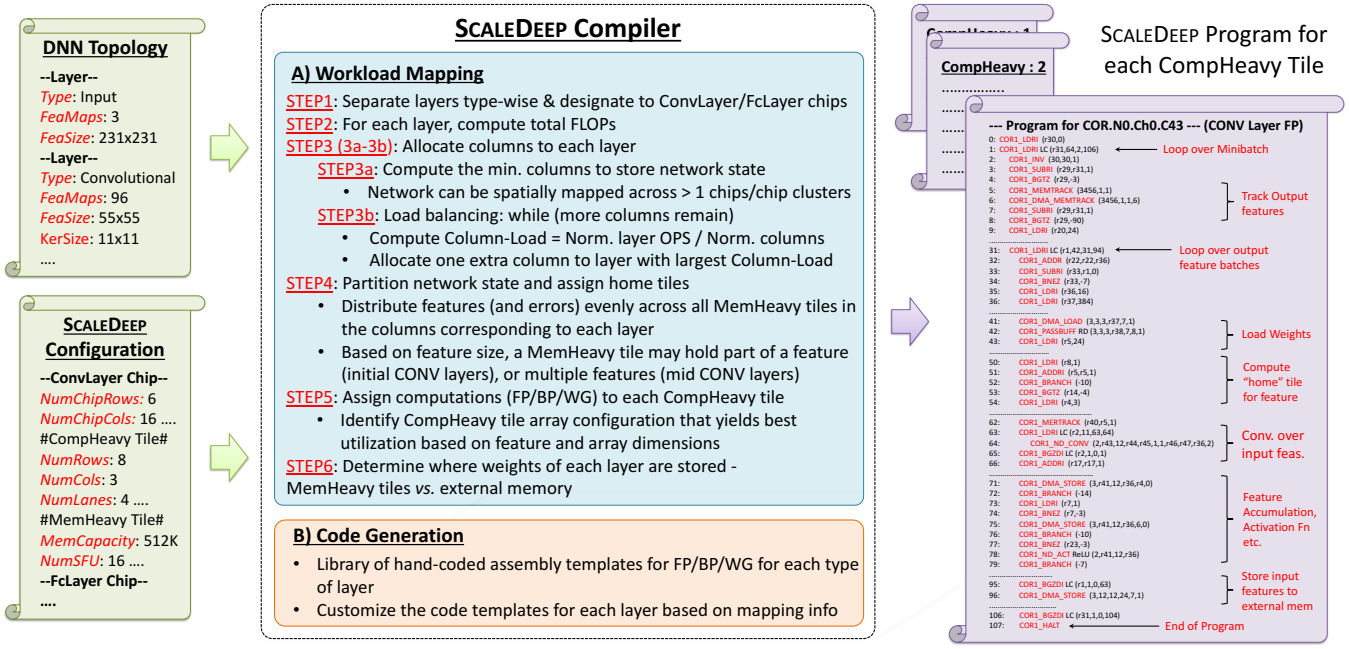


Figure 13: SCALEDEEP Compiler: Phases of operation and sample SCALEDEEP assembly program

Parameter			Value
SCALEDEEP Node			Num. Chip Clusters: 4
Chip Cluster			Num. Chips (ConvLayer/FcLayer): 5 (4/1)
ConvLayer Chip			Num. Chip Rows: 6
			Num. Chip Columns: 16
			Num. CompHeavy/MemHeavy tiles: 288 / 102
	CompHeavy Tile		Array dimensions (Row x Col): 8 x 3
			Lanes / 2D-PE: 4
			Capacity (Left/Top/Bot./Scratchpad): 8K/4K/4K/16K
	MemHeavy Tile		Memory Capacity: 512K
			Num. SFU: 32
			Ext. Mem./Comp-Mem/Mem-Mem Tile BW: 150/24/36 GBps
FcLayer Chip			Num. Chip Rows: 6
			Num. Chip Columns: 8
			Num. CompHeavy/MemHeavy tiles: 144 / 54
	CompHeavy Tile		Array dimensions (Row x Col): 4 x 8
			Lanes / 2D-PE: 1
			Capacity (Left/Top/Bot./Scratchpad): 8K/12K/12K/0K
	MemHeavy Tile		Memory Capacity: 1M
			Num. SFU: 32
			Ext. Mem./Comp-Mem/Mem-Mem Tile BW: 300/48/144GBps
			Wheel Spoke/Arc Bandwidth: 0.5 / 16 GBps
			Ring Bandwidth: 12 GBps
			Operating Frequency: 600 MHz

Power Frac. (Logic, Mem, Interconnect)	
SCALEDEEP	1.4KW (0.5,0.1,0.4)
Chip Cluster	325.6W (0.55,0.1,0.35)
ConvLayer Chip	57.8W (0.7, 0.1, 0.2)
Conv CompHeavy Tile	143.8mW (0.95, 0.05,--)
Conv MemHeavy Tile	47mW (0.3,0.7,--)
FcLayer Chip	15.2W (0.45,0.25,0.3)
Fc CompHeavy Tile	45.9mW (0.95,0.05,--)
Fc MemHeavy Tile	78.6mW (0.2,0.8,--)

	Peak FLOPs (SP)	Processing Efficiency (FLOPs/W)
SCALEDEEP	0.68P	485.7G
Chip Cluster	169.2T	526.5G
ConvLayer Chip	40.7T	703.5G
Conv CompHeavy Tile	134G	934.6G
Conv MemHeavy Tile	19.2G	408.5G
FcLayer Chip	6.6T	432G
Fc CompHeavy Tile	38.4G	836.6G
Fc MemHeavy Tile	19.2G	244.3G

Figure 14: SCALEDEEP: Micro-architectural parameters

SCALEDEEP **Configuration.** Figure 14 shows the micro-architectural parameters of the single-precision SCALEDEEP baseline design. The node contains four chip clusters, each containing 4 ConvLayer and 1 FcLayer chips. The ConvLayer chip contains 6 rows and 16 columns with 288 CompHeavy and 102 MemHeavy tiles. Each CompHeavy tile in the ConvLayer chip has 8 rows and 3 columns, for a total of 24 2D-PEs with 4 lanes each. The MemHeavy tile has 512KB capacity with 32 SFU units. The on-chip links have a bandwidth of 24-36 GBps, while each external memory chip provides 150 GBps bandwidth. On the other hand, the FcLayer chip has 6 rows and 8 columns, with approximately half the number of CompHeavy (144) and MemHeavy (54) tiles. Each CompHeavy tile is also smaller, with array dimensions of 4×8 and one lane/2D-PE. The MemHeavy tile is 1MB in capacity with 32 SFU units. The FcLayer chip's bandwidth is 2×-4× greater than the ConvLayer chip. The bandwidth specifications for the chip cluster/node-level links are also listed in Figure 14. Overall, the SCALEDEEP node contains 5184 CompHeavy and 1848 MemHeavy tiles, and achieves 680 TFLOPS peak at a modest operating frequency of 600 MHz.

Figure 14 also provides a breakdown of the power consumption, peak FLOPs and processing efficiency of the components in SCALEDEEP. Each ConvLayer and FcLayer chip consume ∼58W and ∼15W power respectively. This leads to ∼325W power consumption at the chip cluster level, and 1.4KW for the complete SCALEDEEP node. SCALEDEEP achieves a peak processing efficiency of 485.7 GFLOPs/W, while individual CompHeavy tiles reach up to 934.6 GFLOPs/W.

Benchmarks. To evaluate SCALEDEEP, we utilize 11 state-of-the-art image recognition DNNs listed in Figure 15, as our benchmarks. The benchmark networks include winners from 5 years of the ILSVRC challenge and contain between 11-39 layers (5-33 CONV, 1-3 FC, 3-5 SAMP), 0.65M-14.9M neurons, 6.8M-145.9M weights, and 0.66B-19.4B connections.

6 RESULTS

In this section, we present the results of our experiments that evaluate the benefits of SCALEDEEP.

6.1 Training and Evaluation Performance

Figure 16 shows the performance SCALEDEEP achieves in training and evaluating DNNs, quantified in terms of images per second. The number of chip columns used to spatially realize each DNN is also shown in Figure 16, and ranges between 10-256 depending on the network size. Thus, our benchmark set comprises of scenarios where more than one copy of the network fits on a chip to cases

Benchmark	Layers (CONV/FC/ SAMP)	Neurons (Millions)	Weights (Millions)	Connection (Billions)
AlexNet	11 (5/3/3)	0.65	60.9	0.66
ZF (Clarifai)	11 (5/3/3)	1.51	62.3	1.10
CNN-S	11 (5/3/3)	1.70	80.4	2.57
Overfeat-Fast	11 (5/3/3)	0.82	145.9	2.66
Overfeat-Acc.	12 (6/3/3)	2.05	144.6	5.22
GoogLenet	17 (11/1/5)	2.64	6.8	2.44
VGG-A	16 (8/3/5)	7.43	132.8	7.46
VGG-D	21 (13/3/5)	13.5	138.3	15.3
VGG-E	24 (16/3/5)	14.9	143.6	19.4
ResNet18	23 (17/1/5)	2.31	11.5	1.79
ResNet34	39 (33/1/5)	3.56	21.1	3.64

Figure 15: DNN benchmarks

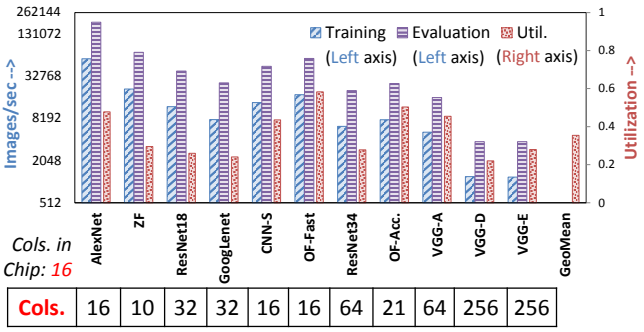


Figure 16: Single precision: Training & evaluation performance

where the network is spread across multiple chips and chip clusters. We observe that SCALEDEEP achieves a training throughput of thousands of images/second across all networks. The evaluation throughput is higher than training by a factor marginally over 3×. This is because, during evaluation, the BP/WG CompHeavy tiles could also be used to perform FP and the overheads such as weight gradient accumulation incurred at the end of each training minibatch do not exist.

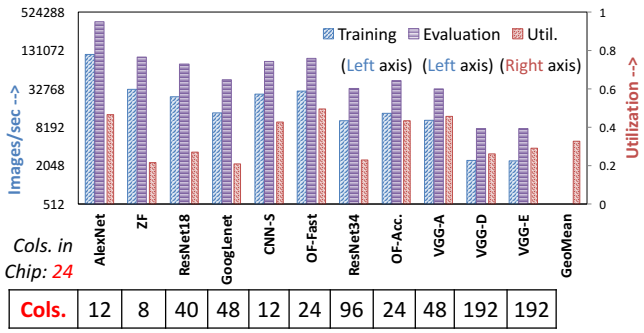


Figure 17: Half precision: Training & evaluation performance

Recent efforts have demonstrated that DNNs achieve state-of-the-art classification accuracy even at lower precisions [25, 50], and therefore we evaluate the performance of SCALEDEEP when all data-structures (features, errors, weights and weight gradients) in the DNN are represented using half precision floating point (FP16). To this end, we replaced each compute unit in the CompHeavy tiles with half-precision floating point units, and reduced the memory capacity of each MemHeavy tile and the bandwidth of each link in the system by a factor of 2. We then varied the micro-architectural parameters to achieve roughly the same power as the single-precision SCALEDEEP design. Specifically, we increased the number of chip rows and columns from 6→8 and 16→24 for the ConvLayer chip, and 6→8 and 8→12 for the FcLayer chip respectively. At half-precision, SCALEDEEP delivers ∼1.35 peta half-precision FLOPS peak, and as shown in Figure 17, achieves 1.85× and 1.82× speedup over SCALEDEEP at single-precision for training and evaluation respectively.

Performance Comparison. In Figure 18, we compare SCALEDEEP to state-of-the-art GPU implementations of popular benchmarks (AlexNet, OverFeat, VGG-A and GoogLeNet) for which data are publicly available [4, 9]. We note that these comparisons were made at the chip cluster level, since commercial GPU cards on which these implementations were realized consume roughly the same power as a chip cluster (∼320W). Overall, a single SCALEDEEP chip cluster achieves 22×-28×, 6×-15× and 7×-11× speedup compared to cuDNN-R2, Nervana Neon and Google Tensorflow implementations on NVIDIA's TitanX GPU. The Titan X GPU used in these results is based on the Maxwell architecture. We were unable to compare directly with the more recent Pascal architecture, as the data for Pascal is not readily available in the public domain. However, comparing Maxwell *vs.* Pascal, the peak single-precision performance improved by ∼1.5× (7 *vs.*11-TFLOPS) [12]. SCALEDEEP outperforms Pascal by 4.6×-7.3× at single-precision, even assuming perfect performance scaling from Maxwell to Pascal. We expect the speedup to be similar at half-precision, given the 1.85× performance scaling between single and half-precision SCALEDEEP designs. Further, recent versions of cuDNN and Neon employ a new convolution algorithm called Winograd [35] to boost performance. Our speedup ranges between 5×-11× compared to Winograd-based GPU implementations. We note that SCALEDEEP implementations currently do not use Winograd, and we do not find any fundamental bottlenecks in doing so to further improve its performance.

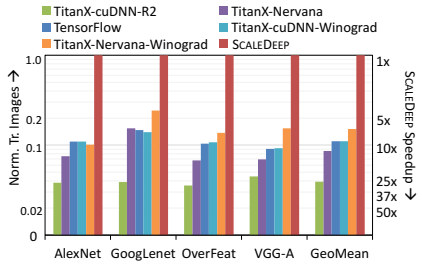
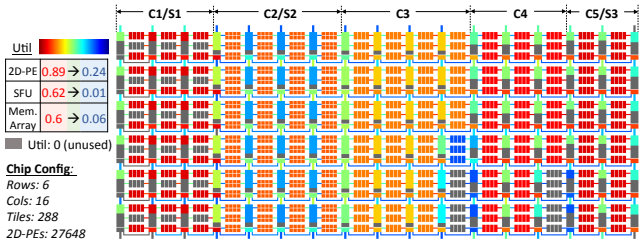


Figure 18: SCALEDEEP **Speedup**

Layer-wise Performance Analysis. Figure 16 also shows the overall utilization of the compute elements in the processing tiles of SCALEDEEP. On an average, we achieve a utilization of 35% across all benchmarks. We quantify the sources of drop in utilization by breaking down the performance layer-wise. Since the network is spatially realized across chip columns and operated as a pipeline, the slowest layer limits the overall throughput. Figure 19 shows the spatial execution of CONV/SAMP layers in AlexNet as an example. The columns allocated to each layer are identified, and the CompHeavy tiles are colored based on the utilization of their 2D-PEs.

For maximum utilization, the 2D-PEs need to be distributed in proportion to the FLOPs in each layer (Row 2 of table in Figure 19). We identify 4 key factors that result in a sub-optimal distribution.

First, for the ease of compilation, we impose a limitation that layers need to be distributed at the granularity of columns. This forces more 2D-PEs to be allocated to some layers, while under-provisioning the rest. In the case of AlexNet, distributing columns as shown in row 3 of the table, makes the number of 2D-PEs allocated to layer C2/S2 lower than ideal, which limits the utilization to 0.74. The next factor stems from the way features are allocated to the MemHeavy tiles. If the layer's feature count is not a multiple of the number of tiles per column, some tiles in the final column may be left with fewer or no features. In our example, layers C3 and C4 have 2 tiles unused, which brings down their respective performance (row 4 of the table). The third factor is intrinsic to each convolution. When the feature sizes are not a multiple of the number of rows in the array, the final iteration leaves some of the rows unused. In some cases, lanes in the 2D-PE may be unused if some iterations do not have sufficient output features. The array reconfigurability was provided to mitigate precisely this issue, and we find that layer C2/S2 leverages this to perform 2 batch convolutions in parallel. Note that the CompHeavy tiles corresponding to C2/S2 are split row-wise in Figure 19. Notwithstanding this, we still find a considerable drop in 2D-PE utilization (row 5 of the table). The final factor is the overhead added due to other program instructions for loop control, data transfer *etc.* This could be reduced through more careful inter-feature pipelining. Across all the benchmarks, we find the 2D-PE utilization to be 0.68 after the column allocation, 0.64 after feature distribution, 0.42 after accounting for array under-use, and 0.35 overall.



FLOPs	0.6G	1.3G	0.9G	0.67G	0.45G
Ideal 2D-PE Dist.	4377	9297	6213	4668	3084
Num. Cols / 2D-PEs / Peak Util.	3 / 5184 / 1.18	4 / 6912 / 0.74	4 / 6912 / 1.11	3 / 5184 / 1.11	2 / 3456 / 1.12
Num. Tiles / 2D-PEs / Peak Util.	18 / 5184 / 1.18	24 / 6912 / 0.74	22 / 6336 / 1.02	16 / 4608 / 0.99	2 / 3456 / 1.12
2D-Array residue Peak Util.	0.55	0.61	0.71	0.71	0.74
Achieved Util.	0.49	**0.48**	0.53	0.61	0.66

Figure 19: Compute and Memory Utilization for AlexNet

We note that most of the above factors are not a fundamental limitation of the SCALEDEEP architecture, but rather a consequence of the current implementation of the SCALEDEEP compiler. For example, the column-level utilization drop can be eliminated if we allow a layer to occupy part of the column, while allocating its remaining tiles to the following layer. We will explore such optimizations to improve the performance of SCALEDEEP as part of future work. Figure 19 also shows the SFU utilization, fraction of cycles where the data array is busy, and the memory capacity that is left unused in each MemHeavy tile. We find that MemHeavy tiles of layers with over-provisioned columns have lower utilization.

6.2 Average Power and Processing Efficiency

Figure 20 shows the average power consumed by the different benchmarks during training, normalized to the peak value. The power consumption attributed to the subsystems—compute, memory and interconnect—is also shown in Figure 20. We find that the compute and interconnect powers scale proportional to the 2D-PE and on/off-chip link utilizations. The memory power, which is largely dominated by leakage, remains largely constant. We correspondingly achieve 331.7 GFLOPs/W efficiency on an average, as listed in Figure 20.

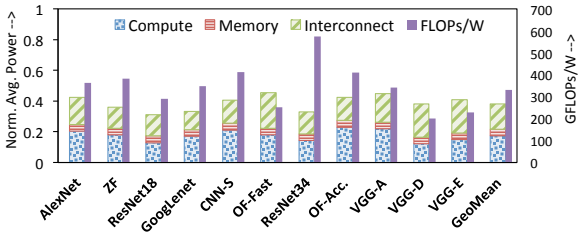


Figure 20: Average power and processing efficiency

6.3 Bandwidth Utilization

Finally, we present the utilization of the on-chip, cluster-level and node-level links for each benchmark during training in Figure 21. In the case of on-chip links, we find the CompHeavy-MemHeavy tile links to be the best utilized (0.87). The traffic through these links are the most predictable as they are confined to carrying the input and output data for each batch convolution/matrix multiplication. On the other hand, traffic through the inter-MemHeavy-tile links is contingent upon factors such as number of columns a layer is mapped to, home row of the features under evaluation, and whether the weights need to be fetched from off-chip. Hence, these links experience lower utilization.

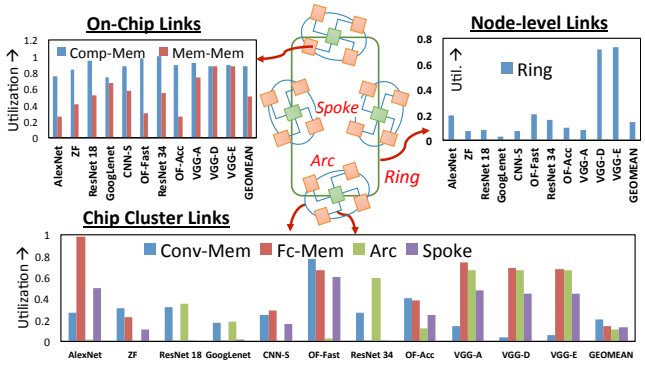


Figure 21: Bandwidth utilization of links in SCALEDEEP

At the cluster-level, the utilization is dictated by the overall throughput and how much the network is spread across the chips. For example, spreading the CONV layers across 2 chips reduces the ConvLayer chips' bandwidth (weights fit on-chip *etc.*), but increases the FcLayer chip's bandwidth, as its batch size decreases by 2×. We find the FcLayer bandwidth utilization to vary significantly across networks. DNNs such as GoogLeNet and ResNet contain a single FC layer with a small number of inputs (in contrast to VGG/OverFeat),

which drastically reduces their bandwidth. DNNs whose CONV layers fit on a single chip have very minimal use for the wheel arcs (only to distribute weights after each minibatch), while spoke utilization is determined by the inputs to the first FC layer. At the node-level, the utilization of the ring is small for all benchmarks except VGG-D/E, which are spatially mapped across multiple chip clusters, requiring the ring to also carry CONV features/errors.

7 RELATED WORK

Prior research efforts to improve the efficiency of DNNs can be grouped into 4 broad directions, as shown in Figure 2. In this section, we discuss these related research efforts and highlight the distinguishing features of our work.

Parallel Software. The most common approach focuses on methods to parallelize DNNs on commercial multi-cores and GPUs [8, 19, 20, 31, 33, 42, 55]. State-of-the-art software implementations on many-core processors (Intel Xeon Phi, NVIDIA Titan) achieve processing speeds in the order of 2-10 TFLOPS with efficiencies of 10s of GFLOPs/W. This translates to time-to-train of several days in the context of large-scale networks. Further, scaling DNNs to clusters with 100s-1000s of nodes is challenging, due to global communication and synchronization bottlenecks. Clearly, over an order of magnitude improvement in performance and efficiency is desired.

Specialized Hardware. A promising approach to improve efficiency of DNNs is hardware specialization. Almost all related efforts [6, 13–15, 17, 22–24, 26, 38, 39, 41, 49] target DNN evaluation and/or one of its key computational kernels (*e.g.* convolution). As described in Section 2, the characteristics of training are quite different, with data flowing in both directions due to BP/WG steps, the need for aggregating gradients after a minibatch *etc.* Also, SCALEDEEP differs in scope; we build a scalable node-level architecture, as opposed to standalone IP cores.

The closest work that shares the objective of SCALEDEEP is DaDianNao [16]. However, SCALEDEEP differs from DaDianNao in several important ways. First, DaDianNao is a homogeneous design, which does not exploit the heterogeneity within and between layers. Specifically, the ratio of compute to memory/interconnect is identical for all tiles in DaDianNao. As the workload characteristics vary by over 100× (between initial-, mid-CONV, and FC layers), it leads to either over-provisioning of memory capacity/bandwidth or under-utilization of the processing cores. Also, DaDianNao adopts a conventional fat tree interconnect topology, which does not leverage the data-flow in DNNs, and incurs additional power and protocol overheads. Quantitatively, SCALEDEEP delivers 5× as many FLOPs as DaDianNao at iso-power. We are unable to directly compare images/second, as the absolute numbers are not presented in [16]. It is also noteworthy that the only full-DNN presented in DaDianNao is AlexNet, the smallest of our benchmarks. SCALEDEEP is the first to demonstrate performance across several DNN topologies.

Approximate Computing. Due to their large-scale and application context, DNNs are robust to impreciseness in selected computations. Design approaches such as low-precision fixed point implementations [18, 25, 50, 54], lossy model compression [27, 32], among others [44, 52], have leveraged this forgiving nature to benefit efficiency, both for training and evaluation.

Post-CMOS Technology. The final direction explores alternate device technologies, wherein the mathematical functions realized by the intrinsic device physics closely resemble the computations present in DNNs. Implementations of small-scale DNNs using memristive crossbars [37, 46] and spintronic devices [40] are representative examples.

8 CONCLUSION

The widespread use of DNNs across the spectrum of computing devices, from mobile devices to the cloud, has ushered in the need to improve their implementation efficiency. Realizing this need, we propose SCALEDEEP, a specialized node architecture for training and evaluating DNNs. The architecture of SCALEDEEP comprises of heterogeneous processing tiles *viz.* CompHeavy tiles and MemHeavy tiles, and heterogeneous compute chips *viz.* ConvLayer chips and FcLayer chips, interconnected using a 3-tiered grid-wheel-ring topology. This leverages the computation and communication patterns prevalent in DNNs. We also develop a compiler that systematically maps any DNN topology to SCALEDEEP. Across 11 DNNs, An embodiment of SCALEDEEP with 7032 processing tiles demonstrates significant performance gains over the state-of-the-art.

REFERENCES

[1] 2013. Improving Photo Search: A Step Across the Semantic Gap. *Google Research blog* (2013).
[2] 2014. Skype Translator - How it Works: http://blogs.skype.com/2014/12/15/skype-translator-how-it-works/. *Skype Blog* (2014).
[3] 2016. Apple is turning Siri into a next-level Artificial Intelligence: http://mashable.com/2016/06/13/siri-sirikit-wwdc2016-analysis/hLMSxZKVnEqO. *Mashable* (2016).
[4] 2016. ConvNet Benchmarks: https://github.com/soumith/convnet-benchmarks. (2016).
[5] 2016. Driver's Ed for Self-Driving Cars: How Our Deep Learning Tech Taught a Car to Drive: https://blogs.nvidia.com/blog/2016/05/06/self-driving-cars-3/. *NVIDIA blog* (2016).
[6] 2016. Google supercharges machine learning tasks with TPU custom chip. *Google Research blog* (2016).
[7] 2016. Introducing DeepText: Facebook's text understanding engine: https://code.facebook.com/posts/181565595577955/introducing-deeptext-facebook-s-text-understanding-engine/. *Facebook Code* (2016).
[8] 2016. Neon, Nervana Systems: http://neon.nervanasys.com/docs/latest/index.html. (2016).
[9] 2016. Nervana Zoo: https://gist.github.com/nervanazoo. (2016).
[10] 2016. Princeton Deep Driving: http://deepdriving.cs.princeton.edu/. (2016).
[11] 2016. Synopsys Design Compiler: http://www.synopsys.com/Tools/ Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx. (2016).
[12] 2016. Titan X: https://blogs.nvidia.com/blog/2016/07/21/titan-x. (2016).
[13] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 1–13. https://doi.org/10.1109/ISCA.2016.11
[14] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A Dynamically Configurable Coprocessor for Convolutional Neural Networks. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 247–257. https://doi.org/10.1145/1816038.1815993
[15] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967
[16] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 609–622. https://doi.org/10.1109/MICRO.2014.58
[17] Y. H. Chen, T. Krishna, J. Emer, and V. Sze. 2016. 14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In

2016 IEEE International Solid-State Circuits Conference (ISSCC). 262–263. https://doi.org/10.1109/ISSCC.2016.7418007

[18] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *CoRR* abs/1511.00363 (2015). http://arxiv.org/abs/1511.00363

[19] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidyanathan, Srinivas Sridharan, Dhiraj D. Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed Deep Learning Using Synchronous Stochastic Gradient Descent. *CoRR* abs/1602.06709 (2016). http://arxiv.org/abs/1602.06709

[20] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25*, P. Bartlett, F.c.n. Pereira, C.j.c. Burges, L. Bottou, and K.q. Weinberger (Eds.). 1232–1240. http://books.nips.cc/papers/files/nips25/NIPS2012_0598.pdf

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

[22] S. Eldridge, A. Waterland, M. Seltzer, J. Appavoo, and A. Joshi. 2015. Towards General-Purpose Neural Network Computing. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 99–112. https://doi.org/10.1109/PACT.2015.21

[23] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *CVPR 2011 WORKSHOPS*. 109–116. https://doi.org/10.1109/CVPRW.2011.5981829

[24] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. 2014. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 696–701. https://doi.org/10.1109/CVPRW.2014.106

[25] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 1737–1746.

[26] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[27] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015). http://arxiv.org/abs/1510.00149

[28] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. 2014. Deep Speech: Scaling up end-to-end speech recognition. *CoRR* abs/1412.5567 (2014). http://arxiv.org/abs/1412.5567

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). http://arxiv.org/abs/1512.03385

[30] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *Signal Processing Magazine* (2012).

[31] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. 2015. FireCaffe: near-linear acceleration of deep neural network training on compute clusters. *CoRR* abs/1511.00175 (2015). http://arxiv.org/abs/1511.00175

[32] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. 2014. Speeding up Convolutional Neural Networks with Low Rank Expansions. *CoRR* abs/1405.3866 (2014). http://arxiv.org/abs/1405.3866

[33] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *CoRR* abs/1404.5997 (2014). http://arxiv.org/abs/1404.5997

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 2012.

[35] Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015). http://arxiv.org/abs/1509.09308

[36] Jiwei Li, Michel Galley, Chris Brockett, Georgios P. Spithourakis, Jianfeng Gao, and Bill Dolan. 2016. A Persona-Based Neural Conversation Model. https://www.microsoft.com/en-us/research/publication/persona-based-neural-conversation-model/

[37] Xiaoxiao Liu, Mengjie Mao, Beiye Liu, Hai Li, Yiran Chen, Boxun Li, Yu Wang, Hao Jiang, Mark Barnell, Qing Wu, and Jianhua Yang. 2015. RENO: A High-efficient Reconfigurable Neuromorphic Computing Accelerator Design. In *Proceedings of the 52Nd Annual Design Automation Conference (DAC '15)*. ACM, New York, NY, USA, Article 66, 6 pages. https://doi.org/10.1145/2744769.2744900

[38] Abhinandan Majumdar, Srihari Cadambi, Michela Becchi, Srimat T. Chakradhar, and Hans Peter Graf. 2012. A Massively Parallel, Energy Efficient Programmable Accelerator for Learning and Classification. *ACM Trans. Archit. Code Optim.* 9, 1, Article 6 (March 2012), 30 pages. https://doi.org/10.1145/2133382.2133388

[39] N. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA '17)*.

[40] S. G. Ramasubramanian, R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan. 2014. SPINDLE: SPINtronic Deep Learning Engine for large-scale neuromorphic computing. In *Low Power Electronics and Design (ISLPED), 2014 IEEE/ACM International Symposium on*. 15–20. https://doi.org/10.1145/2627369.2627625

[41] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-power, Highly-accurate Deep Neural Network Accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 267–278. https://doi.org/10.1109/ISCA.2016.32

[42] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783721

[43] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[44] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs, In Interspeech 2014.

[45] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann Lecun. 2013. Overfeat: Integrated recognition, localization and detection using convolutional networks. *http://arxiv.org/abs/1312.6229* (2013).

[46] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 14–26. https://doi.org/10.1109/ISCA.2016.12

[47] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014). http://arxiv.org/abs/1409.1556

[48] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[49] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality Programmable Vector Processors for Approximate Computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2540708.2540710

[50] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. 2014. AxNN: Energy-efficient Neuromorphic Systems Using Approximate Computing. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design (ISLPED '14)*. ACM, New York, NY, USA, 27–32. https://doi.org/10.1145/2627369.2627613

[51] S. Venugopalan, M. Rohrbach, J. Donahue, R. J. Mooney, T. Darrell, and K. Saenko. 2015. Sequence to Sequence - Video to Text. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. 4534–4542. https://doi.org/10.1109/ICCV.2015.515

[52] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2015. Staleness-aware Async-SGD for Distributed Deep Learning. *CoRR* abs/1511.05950 (2015). http://arxiv.org/abs/1511.05950

[53] Xiang Zhang and Yann LeCun. 2015. Text Understanding from Scratch. *CoRR* abs/1502.01710 (2015). http://arxiv.org/abs/1502.01710

[54] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2016. Trained Ternary Quantization. *CoRR* abs/1612.01064 (2016). http://arxiv.org/abs/1612.01064

[55] Aleksandar Zlateski, Kisuk Lee, and H. Sebastian Seung. 2016. ZNN - A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-core and Many-Core Shared Memory Machines. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. 801–811. https://doi.org/10.1109/IPDPS.2016.119