SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training

Eric Qin[1], Ananda Samajdar[1], Hyoukjun Kwon[1], Vineet Nadella[1], Sudarshan Srinivasan[2], Dipankar Das[2], Bharat Kaul[2] and Tushar Krishna[1]

[1]*Georgia Institute of Technology,* [2]*Intel*

[1]{*ecqin, anandsamajdar, hyoukjun, nadella.vineet*}*@gatech.edu*

[2]{*sudarshan.srinivasan, dipankar.das, bharat.kaul*}*@intel.com*

[1]*tushar@ece.gatech.edu*

***Abstract*—The advent of Deep Learning (DL) has radically transformed the computing industry across the entire spectrum from algorithms to circuits. As myriad application domains embrace DL, it has become synonymous with a genre of workloads across vision, speech, language, recommendations, robotics, and games. The key compute kernel within most DL workloads is general matrix-matrix multiplications (GEMMs), which appears frequently during both the forward pass (inference and training) and backward pass (training). GEMMs are a natural choice for hardware acceleration to speed up training, and have led to 2D systolic architectures like NVIDIA tensor cores and Google Tensor Processing Unit (TPU).**

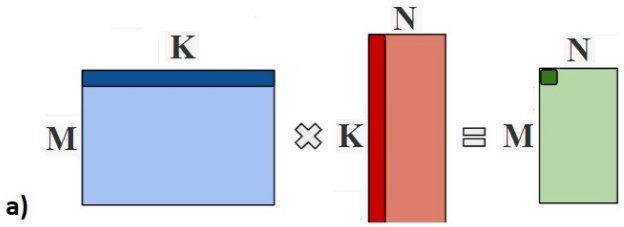
Unfortunately, emerging GEMMs in DL are highly irregular and sparse, which lead to poor data mappings on systolic architectures. This paper proposes SIGMA, a flexible and scalable architecture that offers high utilization of all its processing elements (PEs) regardless of kernel shape and sparsity. Within SIGMA includes a novel reduction tree microarchitecture named Forwarding Adder Network (FAN). SIGMA performs 5.7× better than systolic array architectures for irregular sparse matrices, and roughly 3× better than state-of-the-art sparse accelerators. We demonstrate an instance of SIGMA operating at 10.8 TFLOPS efficiency across arbitrary levels of sparsity, with a 65.10 mm^2 and 22.33 W footprint on a 28 nm process.

I. Introduction

Deep learning (DL) has emerged as the premier algorithmic technique for analyzing data across multiple domains, especially in visual understanding, speech perception, and automated reasoning. The application of DL consists of two steps; namely *training* and *inference*. During training, a Deep Neural Network (DNN) model uses a loss function and optimization process to minimize model error on the training dataset. During inference, the trained DNN model is used to classify data points.

Given latency sensitivity and energy-efficiency demands for DNN inference, a suite of custom accelerators have been proposed [3], [10], [21], [27], [33] to run trained models efficiently by capturing various forms of data reuse [10], [28]. However, *the right acceleration platform for training current and future models*, is still an open question, which is the focus of this work.

The DL training process is extremely compute intensive. This is elucidated in an OpenAI study [6], which shows that the compute requirements for training has grown 300,000 times from AlexNet (2012) to AlphaGo Zero (2018). GPUs are currently the most popular acceleration platform in use for training; and recent research focus on parallelizing large DNN models over multiple GPU nodes. Some companies like Google and Microsoft have built their own specialized training platforms such as the cloud TPU [4] and Brainwave FPGAs [14] respectively.



b)

Workload	Application	Example Dimensions		
		M	N	K
GNMT	Machine Translation	128	2048	4096
		320	3072	4096
		1632	36548	1024
		2048	4096	32
DeepBench	General Workload	1024	16	500000
		35	8457	2560
Transformer	Language Understanding	31999	1024	84
		84	1024	4096
NCF	Collaborative Filtering	2048	1	128
		256	256	2048

Figure 1: **(a) GEMM operation. (b) Example GEMM dimensions from common Deep Learning workloads.**

The core compute component of DL training (and inference) is the GEMM operation [1]. Fig. 1a shows the GEMM dimensions (M, N, K) and operation; while Fig. 1b shows example dimensions found in modern DL workloads. During forward pass, DNNs with fully-connected (FC) layers and multilayer perceptron (MLPs) naturally map to GEMM operations, with MK representing inputs and KN representing weights. For Convolutional Neural Networks (CNNs), GPUs remap the conv operation into a GEMM via the Im2Col operation [18] or other efficient ordering operations. During the backward pass, two GEMM operations arise: MN x $(KN)^T$ and $(MK)^T$ x MN for computing the error gradient w.r.t inputs and weights respectively.

GEMMs comprises around 70% of the total compute cycles during training (as we show in Sec. II); and therefore is a primary target for hardware acceleration. State-of-the-art training accelerators use systolic arrays as the compute fabric for accelerating GEMMs - with sizes ranging from tiny 4x4 engines within each Streaming Multiprocessor in the NVIDIA Volta GPU [31] to a large monolithic 128x128 engine in the Google Cloud TPU [4]. Systolic arrays are built by interconnecting MAC units to a tightly-coupled 2D grid. They are efficient for running GEMMs by enabling reuse of the (M,K) or (K,N) matrix over the rows and columns of the array, reducing data accesses and instruction overheads.

As DNN models evolve at a rapid rate, it is imperative to design the underlying acceleration substrate to remain efficient for future models and training techniques. While GEMMs continue to remain the favorite abstraction to unroll tensors to during the forward and backward passes, architects need to be cognizant of three trends:

- Many GEMMs have irregular (or non-square) dimensions arising from minibatches and weight factorization [34].
- GEMMs exhibit varying degrees of weight sparsity from pruning, and activation sparsity from non-linearities (like ReLU, pooling, and dropout). The number of nonzeros varies throughout training iterations (from 10% to 90%) [48].
- DNN models are being developed at a rapid rate as AI becomes evermore pervasive, making it impractical to pick a specific set of matrix dimensions or sparsity ratios to design accelerators for.

Based on our observations, we recommend three key requirements for future GEMM engines.

- **Flexibility:** GEMM engines should be able to efficiently run matrices of arbitrary dimensions.
- **Sparsity Support:** GEMM engines need support for *unstructured sparsity in both weights and activations* in order to fully utilize hardware resources.
- **Scalability:** GEMM engines need to scale efficiently for integration into different kinds of accelerators. For example, tiny tensor cores in CPUs/ GPUs to large cores in a future TPU.

Unfortunately, state-of-the-art GPUs and TPUs fare poorly on some of the requirements, as we discuss later in Sec. III. GPUs [31] provide some flexibility in terms of tiling irregular GEMMs into 4x4 chunks and mapping over tensor cores, but add complexity for scheduling and accumulation across SMs. TPU, being a large inflexible 128x128 array, can lead to compute under-utilization if the GEMM dimensions do not align with the dimensions of the physical array. GPUs cannot exploit both input and weight sparsity. Even when only one type of sparsity is present, current CUDA libraries require the datatype to be FP32 and the sparse data to be structured. TPU do not natively support sparsity since its rigid internal connectivity and per-cycle systolic dataflow prevent skipping multiplications with at least one operand that is zero. And finally, while systolic arrays scale well due to a regular 2D structure (as is evident from 4x4 versions in GPUs to a 128x128 version in the TPU), larger arrays take proportionally longer to load data and collect final outputs.

In this work, we demonstrate the microarchitecture of a flexible and scalable GEMM accelerator named SIGMA that can handle (a) arbitrary amounts of sparsity, (b) arbitrary irregularity in GEMM dimensions, while guaranteeing close to full compute utilization. SIGMA's key novelty is a highly Flexible Dot Product Engine (Flex-DPE), that can map GEMMs of arbitrary shapes and sparsity distributions via a rich interconnect fabric. Further, Flex-DPE uses tree-based topologies - enabling data loading and collection times of O(1) and O($\log_2 N$) respectively, instead of O($\sqrt{N}$) for an equivalent sized square systolic array. The full SIGMA engine connects multiple Flex-DPEs together via a simple global network-on-chip (NoC). The NoC allocate a cluster of Flex-DPEs for one GEMM. Each cluster is called a Flexible Dot Product Unit (Flex-DPU). SIGMA can thus morph into a large Flex-DPU running one GEMM or into multiple small variable-sized Flex-DPUs running different GEMMs.

Our key contributions are the following:

1) Analysis of modern DL training workloads to make the case for accelerating sparse, irregular GEMMs.
2) A novel accelerator named SIGMA for handling irregular and unstructured sparse GEMM operations.
3) A novel reduction network, named Forwarding Adder Network (FAN), for efficient partial sum accumulation.
4) Layout implementation of SIGMA for scalability and backend analysis.

The rest of the paper is organized as follows: Sec. II discusses modern training workloads and their GEMM characteristics. Sec. III dissects state-of-the-art deep learning accelerators and design considerations. Sec. IV proposes the SIGMA microarchitecture, and Sec. V describes the physical implementation and hardware costs. Sec. VI evaluates the performance of SIGMA against the state-of-the-art. Sec. VII discusses the related works, and Sec. VIII concludes.

II. DL Training Characterization

In this section, we analyze GEMM kernel shapes and sparsity levels from modern deep learning applications.

Target Workloads. For the kernel characterization exercise, we consider three workloads: Transformer [42], Google Neural Machine Translation (GNMT) [45], and Neural Collaborative Filtering (NCF) [20]. We also leverage "Baidu DeepBench" [2], which identifies key GEMM kernels encountered across various CNNs/ RNNs/ LSTMs. For Transformer, we use a 324 Million parameter model [43]

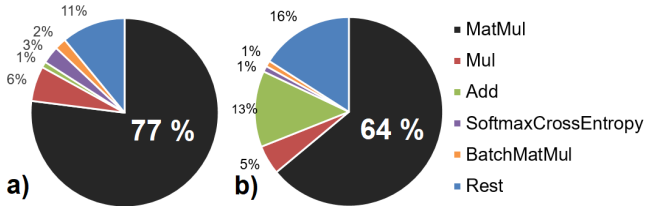


Figure 2: **Time breakdown for different ops on V100 for (a) Transformer and (b) GNMT. Matrix Multiply consume around 70% of total runtime.**

with the LM1B (billion word corpus) dataset. For GNMT, we evaluate the state of art 8-layer GNMT model with WMT-German-English dataset.

Time-Breakdown of Compute Primitives. Figure 2 shows the time break-up of different operations when training GNMT and Transformer on a NVIDIA V100 GPU [31]. We observe that approximately 70% of time is spent on matrix multiplications (MatMul) operations or operations that can cast as MatMuls. ***Thus, MatMul is a key compute primitive to accelerate in hardware to speed-up training.***

GEMM shapes. Transformer, GNMT, NCF and Deep-Bench [2] have matrices of different sizes and shapes as shown in Fig. 1b. Training is performed in different batch sizes, which lead to different input matrix dimensions. The observed shapes of the operand matrices vary from tall-skinny (rows dominate over columns) to fat-short (columns dominate over rows) - this is due to low minibatch sizes. ***Thus, GEMM accelerators need scalability and flexibility to handle large and irregular GEMM sizes efficiently.***

Sparsity within GEMMs. As the objective of this work is not focused on algorithm techniques to generate sparse models, we leverage a pruning approach similar to Zhu et al. [48] via a slow sparsification technique that increases the sparsity level of weights from zero to a final sparsity level in a fixed set of pruning steps.

For GNMT [45] with ∼210M parameters, we achieve close to state-of-the-art accuracy with 90% weight sparsity (resulting in ∼22M parameters), similar to results outlined in [48]. The pruning is applied to embedding, decoder projection layer and all LSTM layers in both the encoder and decoder. Workloads like transformer and ResNet-50 also exhibits good accuracy with around 80% and 70% weight sparsity respectively [15]. Activation sparsity in DNN models comes from ReLU and dropout layers.

Improper handling of sparse matrices wastes compute resources and causes unnecessary but expensive movement of zeros across the memory hierarchy. As matrices are getting bigger and sparser, the need for sparsity support becomes more important. ***Thus, GEMM accelerators need support to handle both weight and activation sparsity efficiently.***

III. Inefficiency of GPUs and TPUs

In this section, we demonstrate the inefficiencies with current GEMM accelerators, and discuss the design choices

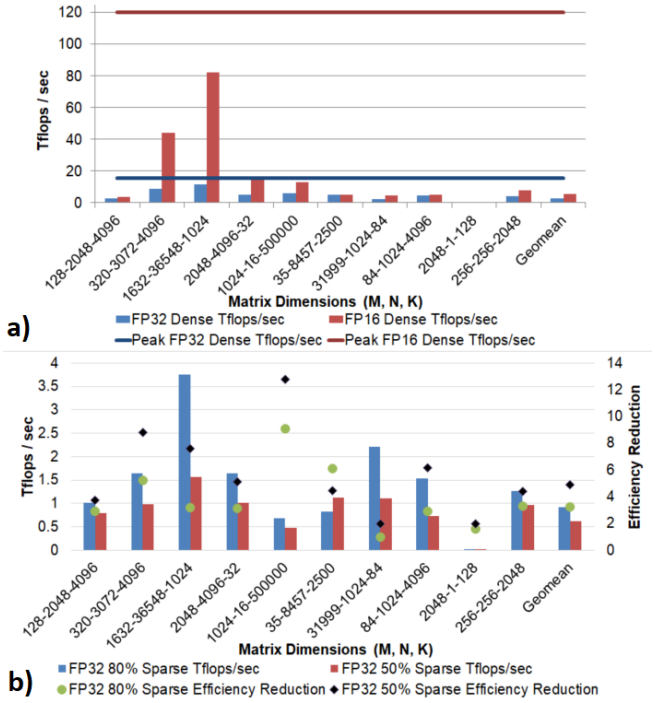


Figure 3: **GPU performance evaluation on different GEMMs.**

that eventually lead to our proposed design.

A. Irregular and Sparse GEMMs on GPU

We measured the compute efficiency on V100 GPUs with and without sparsity for various GEMM dimensions. In Fig. 3a, we run some of the deep learning MatMul kernels (dense irregular without any sparsity) for workloads described in Sec. II on a single card V100 GPU and measure the efficiency with FP32 and FP16 data type. FP16 data type can take advantage of the systolic arrays ("tensor cores") in V100 for GEMM computation. While FP16 uses the tensor cores to boost the efficiency compared to the FP32 version, they still operate at a fraction of the peak efficiency due to irregularity in kernel dimensions; whereas a dense regular GEMM (2k, 2k, 2k) with FP16 tensor cores provide up to 76% efficiency.

We then introduce sparsity to the above MatMul kernels and use NVIDIA cuSPARSE [5] libraries, which support sparse matrices computation. cuSPARSE libraries API currently support only one of the matrices to be sparse with only FP32 data type. In this experiment, we induce random sparsity of 50% and 80% to one of the matrices while keeping the other matrix dense. From Fig. 3b, we observe on average 4x reduction in efficiency compared to the equivalent dense FP32 matrix computation by adding sparsity. We expect the efficiency to decrease further when both matrices are sparse. Current GPU systems cannot efficiently map sparse GEMM computation onto their compute engine when there is no structure in the sparsity, and thus we need to fundamentally re-architect how we design a system that can take advantage

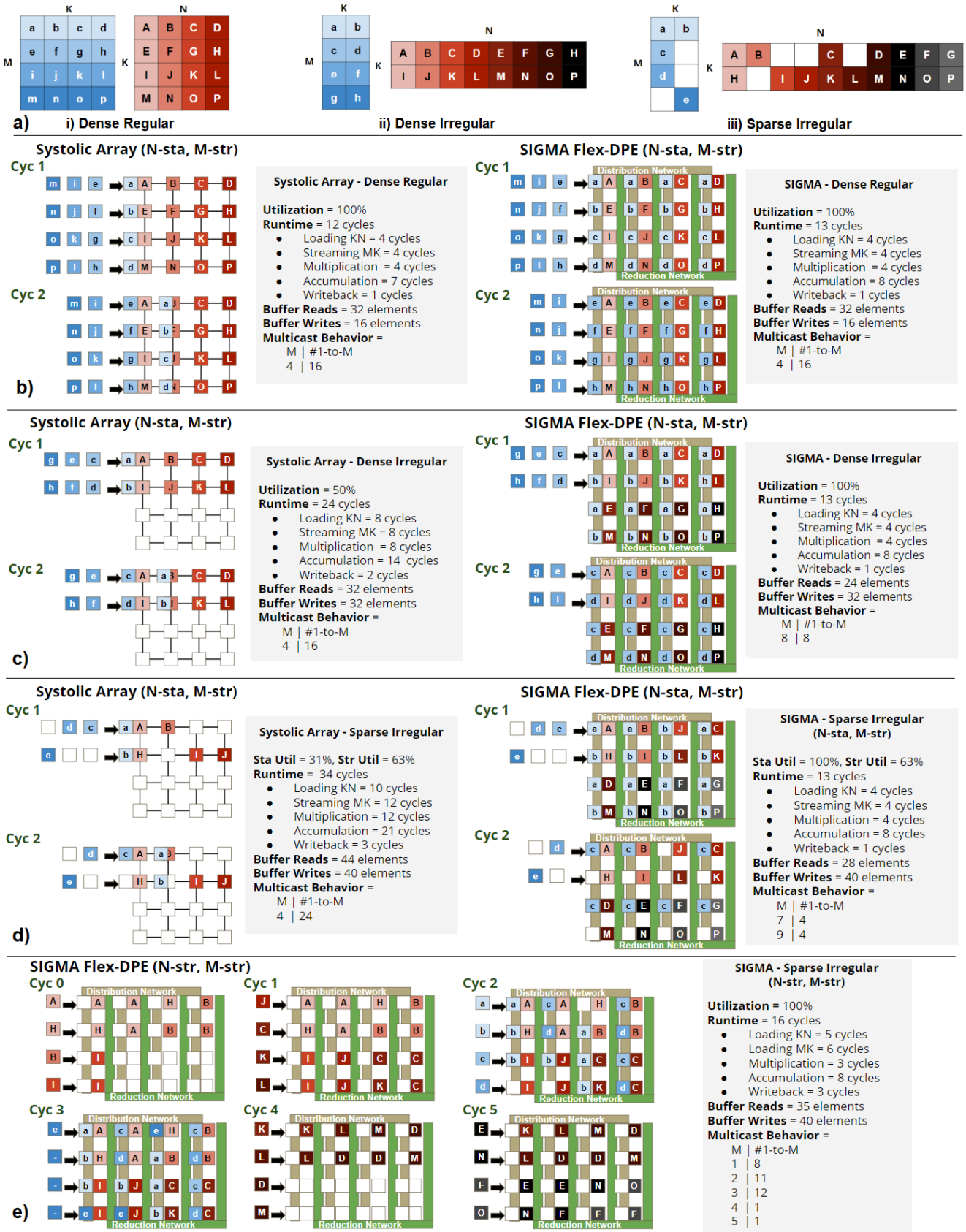


Figure 4: **(a) Example GEMM matrices. (b) Mapping comparison between systolic array and SIGMA Flex-DPE for dense regular GEMMs. (c) Dense irregular GEMMs comparison. (d) Sparse irregular GEMMs comparison. (e) Alternative sparse irregular Flex-DPE mapping. (Note: M-str means MK matrix is streaming, N-sta, means KN matrix is stationary, etc.)**

Feature	SIGMA Goals	Systolic Array Limitation	SIGMA Structure	SIGMA Structure Property
Flexibility	Efficiently map irregular matrices	Rigid aspect ratio	Flex-DPE	1D substrate with ability to mimic any 2D aspect ratio via non-blocking distribution
Sparsity	Weight and activation sparsity	Data forwarding every cycle in horizontal/ vertical directions	Distribution network with data-dependent routing	Only non-zeros distributed. Variable sized simultaneous dot-products.
Scalability	Map large GEMMs	Data distribution and reduction scales as O($\sqrt{N}$)	Benes distribution and spatial reduction network	O(1) distribution and O(log_2N) reduction time.

Table I: **Desired features for a GEMM accelerator, limitations of systolic arrays, and SIGMA's approach.**

of sparse computation to achieve high efficiency for deep learning workloads.

B. Irregular and Sparse GEMMs on TPU

Google's TPUs are a poster-child for large GEMMs due to their 128×128 systolic array. However, across a suite of GEMMs from modern DL workloads, we observe that it is common to have less than 50% of the array utilized when running irregular matrices, as we show later in Sec. VI. In addition, systolic arrays cannot inherently address sparsity. The reasons for these inefficiencies are discussed next.

C. GEMMs on Systolic Arrays vs. SIGMA

Systolic arrays face under-utilization in many different scenarios due to two inherent features: a rigid shape, and a simple but rigid interconnect. In Fig. 4, we contrast a systolic array against an abstract view of SIGMA's Flex DPE (which will be presented in detail later in Sec. IV-A). Fig. 4a shows three example GEMM operations: (i) dense regular, (ii) dense irregular and (iii) sparse irregular matrices. The shaded boxes in the figure highlight quantitative metrics such as utilization, runtime, multicast behavior, and SRAM reads/writes for each example. For the purposes of this example, it is sufficient to assume that SIGMA's Flex-DPE has two specialized networks between the PEs and the SRAMs on the sides of the array (not shown in the figure) - a *distribution network* and a *reduction network*. The specific implementation of these networks is discussed later in Sec. IV.

Dense Regular Matrices. Fig. 4b shows how the dense regular matrices are mapped. In the example, we use a KN matrix stationary, and MK matrix streaming dataflow (N-sta, M-str). An alternate term for this dataflow is *weight-stationary*, and is used by the Google TPU [23], [37]. Partial sums are generated at each cross-point and accumulated over the columns. The systolic array and Flex-DPE designs are able to fully utilize its PEs by mapping all of KN matrix onto its PEs. They key differences between the two are that (i) a systolic array sends the streaming matrix in a store and forward manner, while SIGMA multicasts the data to the corresponding PEs in one cycle, and (ii) the systolic array uses a linear reduction while SIGMA uses a tree-based reduction, as we describe later in Sec. IV.

Dense Irregular Matrices. Fig. 4c shows how dense irregular matrices are mapped. A systolic array design suffers from under-utilization due to its rigid structure. Despite the fact that there are 16 elements in the dense irregular KN matrix and 16 PEs in the systolic array, only half of the matrix can be mapped at a time. *This is due to the rigid shape of the systolic array*. All partial sums are accumulated down a column via forwarding; mapping the other half of the dense irregular N-matrix onto the systolic array at the same time will lead to incorrect functionality, since the accumulated output (a.A+b.I) should not get added to (a.E + b.M). The second half of the N-matrix will therefore have to be loaded once the first half of the matrix is computed, more than doubling the computation time. In contrast, the Flex-DPE is able to map all of the dense irregular stationary elements at one go, utilizing all PEs completely. This is enabled by having a flexible reduction network that can accumulate both sets of outputs (a.A+b.I) and (a.E + b.M) separately and concurrently, as we describe later in Sec. IV. This not only provides a runtime advantage, but also an energy advantage since the M-matrix only needs to be read and streamed through the array once.

Sparse Irregular Matrices. Fig. 4d shows how sparse irregular matrices are mapped. Not only does a systolic array suffer under-utilization from irregular matrices, but also from sparsity. To maintain correctness of the final output, a systolic array must map the non-zero values onto the compute unit. *This limitation comes due to the rigid forwarding network between PEs.* The Flex-DPE design is able to map only non-zero elements because of the flexible distribution and reduction networks. There are two different dataflows enabling sparse compute in a Flex-DPE. The N-sta, M-str dataflow for Flex-DPE in Fig. 4d maps only non-zero elements onto the compute, giving it 100% stationary utilization, making it more efficient than the systolic array. However, the streaming matrix may send zero-valued elements. This is because all non-zero stationary elements are mapped if there is at least one streaming value that needs to be multiplied with it.

Fig. 4e shows the N-str, M-str dataflow (i.e., No Local Reuse [10]) for the Flex-DPE that fully utilizes the compute. This is done by streaming only necessary multiplication pairs and not keeping any values stationary. We provide more details about the distribution and reduction network architecture that can enable this feature in Section IV. The equivalent dataflow is not possible for the systolic array as it does not allow arbitrary pairings of vectors from the M and N matrices due to its rigid cycle-by-cycle forwarding network.

Distribution and Reduction Latency. Another point to notice from the quantitative data in Fig. 4b-e is that the

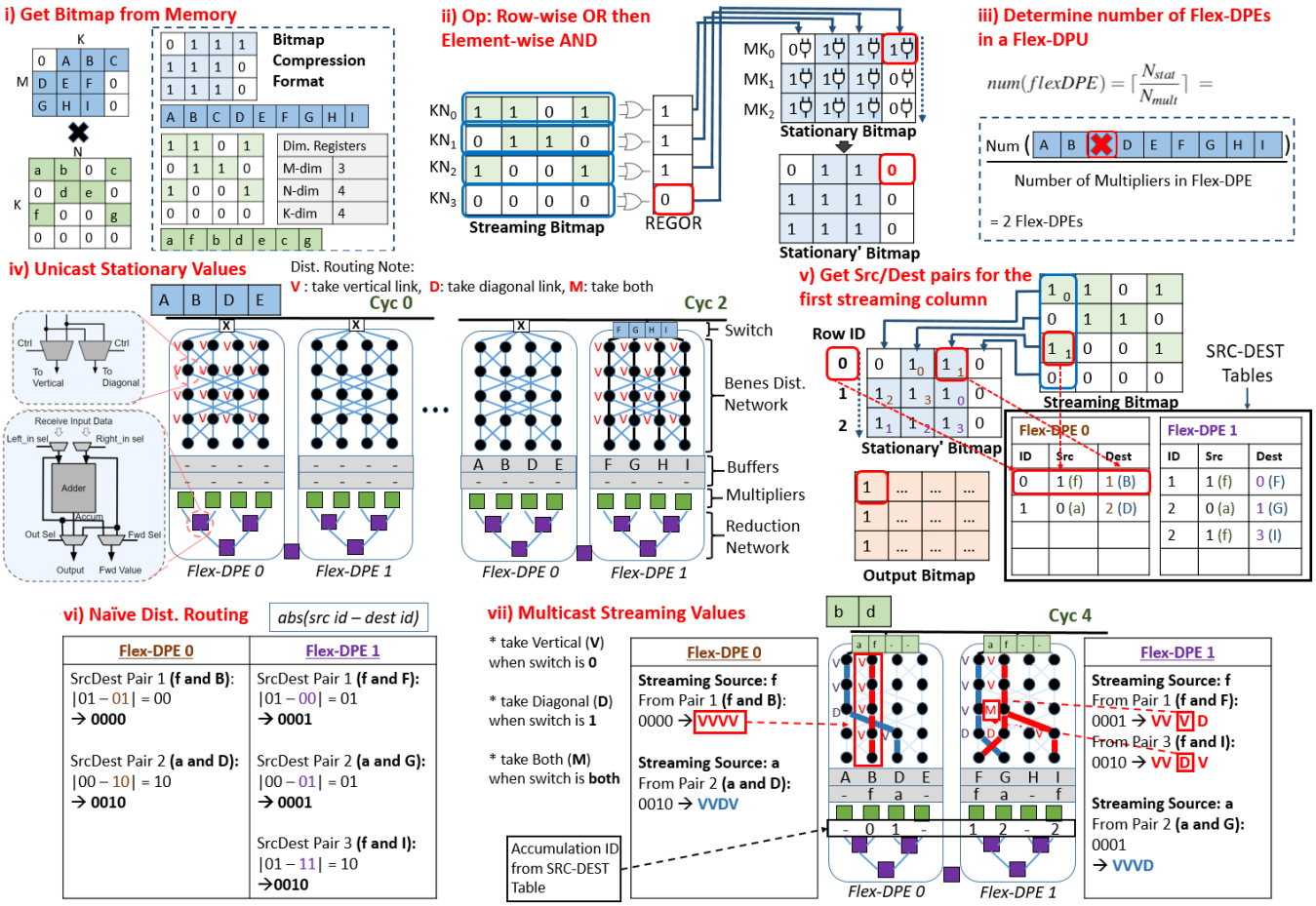


Figure 5: **Walk through from bitmap to SIGMA mapping. This example uses M-stationary, N-streaming dataflow (see Fig. 4d).**

data loading and accumulation time in systolic arrays is always proportional to the array dimensions, while SIGMA's networks allow O(1) distribution and O($\log_2$N) reduction.

Summary. Table I summarizes the sources of inefficiency in systolic arrays and how SIGMA addresses each. Architectural details are provided next.

IV. SIGMA Architecture

The fundamental building block within SIGMA's compute fabric is a processor named Flexible Dot Product Engine (Flex-DPE), described in Sec. IV-A. Several Flex-DPEs are connected together via a simple NoC to create the full SIGMA compute fabric. Each GEMM operation reserves a contiguous group of Flex-DPEs, creating a Flexible Dot Product Unit (Flex-DPU), described in Sec. IV-B. The memory-system is similar to the TPU [4], [23]. Fig. 8 depicts the high level schematic of SIGMA.

A. Microarchitecture of Flex-DPE

A k-sized Flex-DPE houses k multipliers, $k-1$ adders, local buffers, control unit, and flexible interconnects. The multipliers are laid out in a logical 1-D structure. Fig. 5 shows an overview. A 1D substrate enables us to run matrix-matrix (M*M) multiplications as multiple vector matrix multiplications (V*M), similar to Brainwave [14]. Recall from Fig. 4 that a k-sized square systolic array has $\sqrt{k}$ columns and $\sqrt{k}$ rows, with each column computing an independent dot-product when running a weight [23] or input-stationary dataflow [37]. In contrast, a k-sized Flex-DPE can be configured to create myriad combinations of dot-products: one dot-product of size k, two dot-products of size $k/2$, $\sqrt{k}$ dot-products of size $\sqrt{k}$ (like the systolic array), and so on. In fact, the flexible distribution and reduction networks also enable creation of multiple variable-sized dot-products, which is crucial for sparsity. In Sec. V, we study how the Flex-DPE scales with area and power.

1) Distribution Network: Benes Topology: The role of a distribution network within any GEMM engine is to load the stationary matrix (MN or KN), and stream the other matrix, as shown in Fig. 4. In a systolic array, the distribution network behavior is implemented via the horizontal and vertical forwarding links between PEs. This leads to an O(k) data loading time for a $k \times k$ systolic array.

In SIGMA, we adopt a Benes network [7] to support the flexibility demonstrated in Fig. 4. Benes is a non-blocking N-input N-output multistage network with 2log(N)+1 levels, each with N tiny 2x2 switches. The switches, as shown in

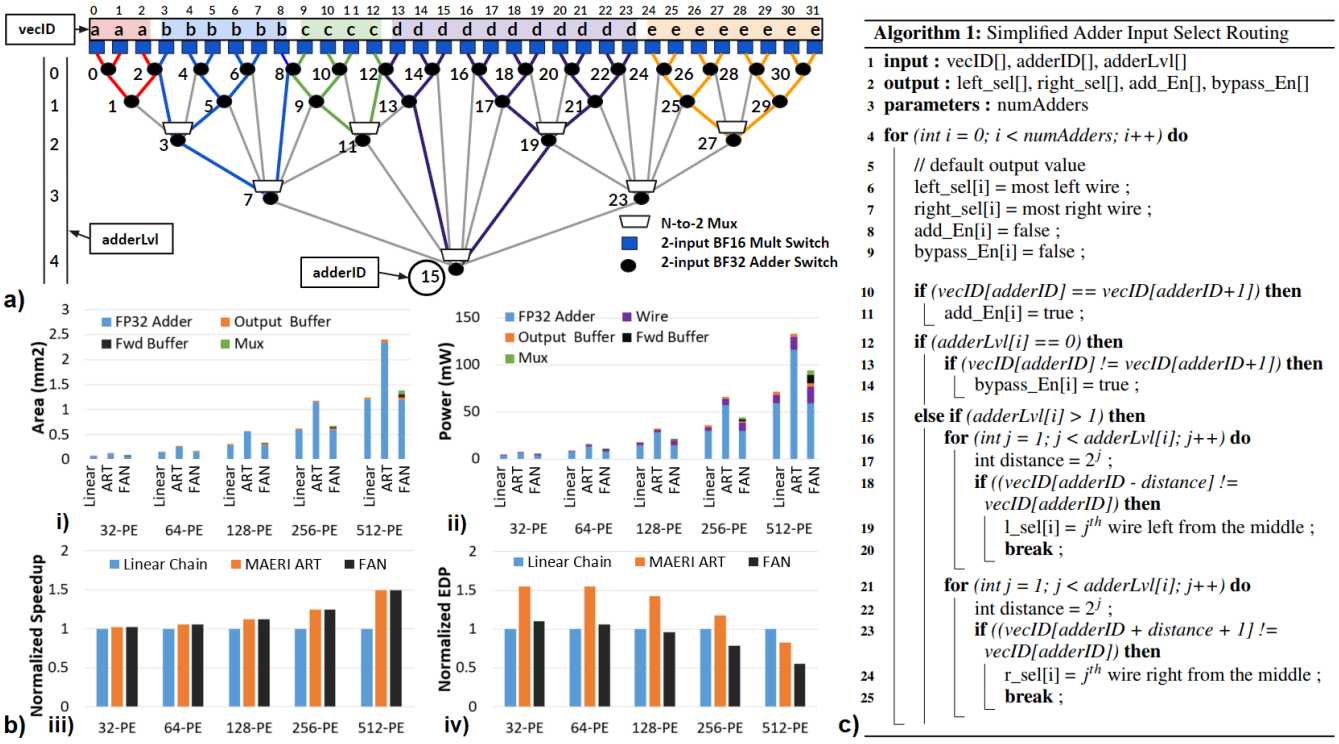


Figure 6: **(a) Fowarding Adder Network Topology. (Not shown are flip flops across each stage of the forwarding links). (b) Spatial FP32 reduction interconnect comparisons of i) post-layout area, ii) post-layout power, iii) speedup, and iv) Energy-delay Product (EDP). (c) Simplified FAN routing algorithm for adder inputs and functionality (Note: variables i and adderID are equivalent).**

Fig. 5-Step(iv), require two control bits; one for selecting the vertical output and one for diagonal output. Numerous Benes routing algorithms have been proposed [7], [8], [29]. The non-blocking property of Benes allows any source to communicate with any destination without any intermediate contention. We use latch-free switches (except for timing closure) at each-stage, allowing a O(1) data communication across the Benes network. We also support multicasts within the Benes network to avoid having to read the same data element multiple times from SRAM.

Other design-choices are also feasible for the distribution network. A crossbar gives the same non-blocking behavior as Benes and has much simpler routing, but it does not scale well (N^2). Blocking interconnects such as buses [10], trees [11], [27], butterfly and mesh networks [9], are still valid design choices due to their low wire costs, but will cause performance degradation due to increased distribution delays.

2) Reduction Network: FAN Topology: Dot-product reduction can be implemented in three ways.

Spatio-Temporal Reduction: The TPU weight-stationary systolic array implementation performs reduction via forwarding along the column, requiring O(k)-cycles for a $k \times k$ array. The time taken is independent of the actual size m of the dot-product which may be smaller.

Temporal Reduction: Designs like EIE [19] perform in-place reduction within PEs. The time taken is still linear like spatio-temporal, but equal to O(m) - i.e., the dot-product size.

Spatial Reduction: In SIGMA, we implement a spatial tree-based reduction, as it requires O($\log_2 m$) cycles, for a m-sized dot-product. The challenge with realizing this $\log_2 m$-cycle reduction, however, is that non-powers of two sized reductions are hard to map over traditional binary adder trees. Suppose we are trying to accumulate three separate dot-products for (*a0, a1, a2, b0, b1, c0, c1, c2*) on an eight-wide adder tree. Following the natural binary-tree topology, *a2-b0* and *b1-c0* will reach the same adder as they go up the tree, which is incorrect functionally.

FAN Topology. To address this issue, we propose a novel adder-tree topology named Forwarding Adder Network (FAN) that places forwarding links between different levels of adders over a traditional binary adder tree. The topology and variable labels of a 32-wide FAN are shown in Fig. 6a. VecIDs and adderIDs are numbered in increasing order from left to right, and each adderID has a corresponding adderLvl value. Below is a pseudocode describing the link connections between adders to create FAN of any power of 2 size.

```
// Adders at level 0 connect to vecID directly
// Adder links start from level 1

for (int i = 0; i < numAdders; i++) do
   int adderID = i; // they are the same
   for (int lvl = 1; lvl <= adderLVL[i]; lvl++) do
      connect with adder: adderID - 2^(lvl-1);
      connect with adder: adderID + 2^(lvl-1);
```

Routing Algorithm. The routing algorithm for FAN is shown in Fig. 6c. For every adder, if vecID[adderID] equals to vecID[adderID+1], accumulation is enabled. If the vecIDs are not equal and the adder is in the zeroth level, the bypass link is enabled. For example, in Fig. 6a, Adder 12 needs to bypass *'c'* and *'d'* to the next adder levels. From the second adder level onward, there is a N-to-2 mux before every FP32 Adder. To determine which inputs get selected, comparators are used to identify cluster regions.

Benefits and Overhead. FAN offers similar benefits as the ART topology proposed in MAERI [27] in terms of creating dot-products of variable sizes. However, FAN is much more lightweight. This is because MAERI's ART is built using three input adders (two from parent nodes, one from a sibling node), which makes it extremely prohibitive, especially when working with FP32 data type (commonly used during DNN training). Fig. 6b shows the performance evaluation between linear reduction (i.e., temporal or spatio-temporal), ART, and FAN. For performance calculations, we use 100 stationary folds (when stationary elements need to be replaced) with stream dimension of 1000 each. As shown in Fig. 6b-iii, taking log$N$ cycles rather than N cycles before starting the next fold significantly improves performance as the number of PEs increases. Our findings show that 512-PE FAN only has a 10% and 31% area power overhead over linear, compared to ART which has a 92% and 86% overhead respectively. FAN also provides EDP benefits over linear starting from 128-PE. At 512-PE, FAN's EDP is 45% and 34% lower than linear and ART respectively. From our results, we conclude that FAN is both high performance and scalable.

3) Execution within Flex-DPE: The Flex-DPE design allows mapping for dense or sparse, and regular or irregular GEMMs. In Fig. 4, we have seen how different combinations of matrices are mapped onto Flex-DPE. Fig. 5 depicts the steps involved in generating the mapping for sparse matrices which we will describe later.

B. Composing Flex-DPEs into a Flex-DPU using a NoC

To extract maximum possible parallelism from the available multipliers, SIGMA can dynamically compose a number of Flex-DPE units together to form a logical ensemble which we call Flex-DPU. A Flex-DPU is responsible for running one GEMM. Multiple Flex-DPUs can be scheduled in parallel to run multiple GEMMs. The NoC connecting the Flex-DPEs together is similar to that of tiled accelerator architectures, such as Tangram [16] and Simba [39].

A simple switch is present at the intersection of each Flex-DPE to arbitrate the flow of the data. These switches are connected together in a 2D mesh. They are statically configured when mapping the GEMMs, and do not require any dynamic routing or flow-control like conventional NoCs. ***The amount of bandwidth on this NoC (i.e., number of unique elements of the row/column that can be transferred per-cycle) is a design-time configurable parameter.***

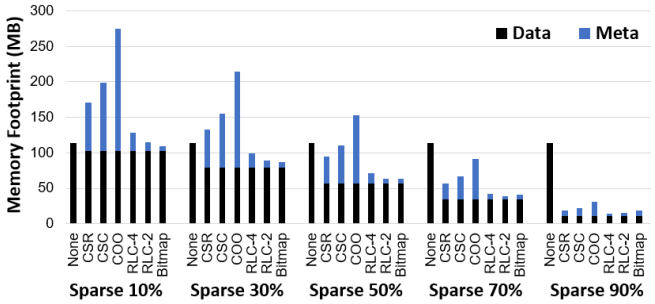


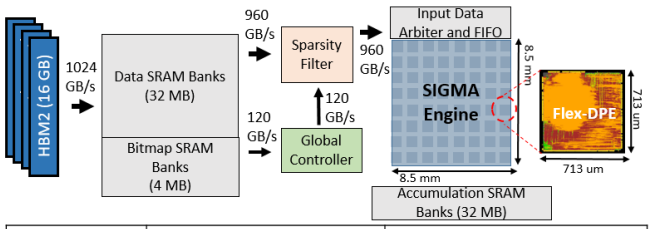
Figure 7: **Matrix memory overhead with dimensions M=1632 and K=36548. Format comparisons include: None, CSR, CSC, COO, RLC-4, RLC-2, and Bitmap in the following order.**

Within a Flex-DPU, the switch forwards data across Flex-DPEs, providing seemless multicasts of data like a bus. We describe this with an example in Sec. IV-E. Across Flex-DPUs, the switches provide hop-by-hop data forwarding, similar conventional NoCs.

C. Supporting Unstructured Sparsity

Compression Format. One of the key benefits of supporting sparsity is low-memory footprint; and consequently more energy savings by avoiding zero-valued element transfers. There are a few well recognized compression formats such as CSC, CSR, COO, and RLC (Run-length compression). We use a Bitmap format within SIGMA, where each element has a corresponding bit to indicate if a given element is zero or non-zero in the corresponding matrix [17], [24], [35], [36]. Fig. 7 compares the metadata overhead of various compression formats with varying levels of sparsity. The dimensions and sparsity levels in the plot reflect what we observe in our workloads (see Sec. II). The metadata overhead for COO/ CSR/ CSC changes drastically at various sparsity regions. This is because each nonzero element require indices of $log_2(dimension)$ bits, etc. The Bitmap format has a constant meta-data overhead irrespective of sparsity, making it attractive for SIGMA which targets arbitrary unstructured sparsity. At low-levels of sparsity, we find Bitmap having lower footprint than COO/ CSR/ CSC. Bitmap has comparable overhead to RLC [10], [19], at sparse ratio of $\sim$30% to $\sim$70%. We observe that RLC is better at reducing meta-data over Bitmap at $>\sim$70% sparsity, but is worse at $<\sim$30% sparsity. We evaluate RLC using 4-bit (RLC-4) and 2-bit (RLC-2) run lengths. *Alternate compression formats can be supported over SIGMA by only changing the front end controller to ensure proper mapping.*

Sparsity Controller. For each GEMM, a global controller determines how sparse matrices are decoded and mapped onto SIGMA. The controller operates on the bitmap meta-data and calculates how many Flex-DPEs are needed. Internal counters and tables are implemented to determine the indices where dense computations are needed. We describe the details with a walkthrough example in Sec. IV-E.



	TPU-like Systolic Engine		SIGMA Engine	
Technology	Commercial 28nm		Commercial 28nm	
Clock freq.	500 MHz		500 MHz	
MACs	16384 (128 x 128 PEs)		16384 (128, 128PEs Flex-DPEs)	
Data Type	BFP16 Mult, FP32 Add		BFP16 Mult, FP32 Add	
Peak TFLOPS	16		16	
Sparsity Support?	No		Yes	
***Effective TFLOPS**	1.88		10.8	
Power (W)	12.25 W		22.33 W	
Eff. TFLOPS/ W	0.15 Eff. TFLOPS/W		0.48 Eff. TFLOPS/W	
Area (mm2)	Total: 47.27 mm2		Total: 65.10 mm2	
	Adder:	14.5 %	Adder:	10.5 %
	Multipliers:	59.0 %	Multipliers:	42.5 %
	Local Memory:	1.5 %	Local Memory:	1.0 %
	Layout Overhead:	25.0 %	Dist. NoC Overhead:	14.5 %
			Red. NoC Overhead:	3.0 %
			FAN Controller:	1.5 %
			Layout Overhead:	27.0 %

Figure 8: **SIGMA high level diagram and comparison table aginst a TPU-like design. We report only the area-power of the compute array, not the SRAMs. Effective TFLOPs is calculated by multiplying the base dense TFLOPs with the average efficiency computed across GEMMs in Sec. VI.**

D. Dataflows Supported by SIGMA

SIGMA's flexible substrate enables it to support myriad dataflows. For all workloads, we use both Weight (i.e., KN) stationary and Input (i.e., MK) stationary dataflows (Fig. 4d), and pick the one that provides higher efficiency. In these two dataflows, spatial dot-products of dynamic size are created depending on the matrix dimensions and sparsity of the stationary matrix. The columns/rows of the streaming matrix are **reused spatially** by broadcasting to the rows/columns of the stationary matrix (which are **reused temporally** at each multiplier). SIGMA can also run a No Local Reuse (i.e., MN-str, KN-str dataflow from Fig. 4e). This dataflow can provide 100% utilization of the compute, but comes at the cost of requiring higher interconnect bandwidth.

E. Walkthrough Example

The following steps (corresponding to Fig. 5) depicts a walk-through example showing how SIGMA utilizes the bitmap format to map sparse GEMMs onto Flex-DPEs. Here, the number of *multipliers per Flex-DPE (N_{mult})* is four.

- *Step i)* Gather two bitmap-compressed matrices. In this example, MK is stationary and KN is streaming.
- *Step ii)* Conduct row-wise OR across the streaming bitmap and store the outputs to REGOR (temporary registers). Then, do element-wise AND between the corresponding REGOR row and stationary bitmap column to generate stationary' bitmap.
- *Step iii)* The number of ones in stationary' bitmap corresponds to the *number of useful stationary values (N_{stat})*. Since N_{stat} is 8 and N_{mult} is 4 in this example, 2-Flex-DPE units are needed to form a single Flex-DPU.
- *Step iv)* Unicast the stationary values to the multiplier buffers. The routing is straightforward, as all stationary input data travel vertically down. In this example, the input bus has a 4X bandwidth, so it is only able to fill one Flex-DPE each cycle.
- *Step v)* To generate source and destination pairs for each Flex-DPE, a counter value is assigned to each non-zero element in the stationary' and streaming bitmaps. For stationary' bitmap, the counter starts at 0 and increments from left-right, top-bottom. The counter resets when it reaches N_{mult} -1, which marks the end of one Flex-DPE. Counter values increments top-bottom in the streaming bitmap and resets at the start of each column. Then, a streaming bitmap column compares to each row of the corresponding stationary' bitmap. If both values are 1, the counter values are stored in the Flex-DPE SRC-DEST tables. The row-id is recorded to determine partial sum regions. Additionally, an initial output bitmap is generated based on if there are any non-zero computations.
- *Step vi)* Generate distribution routing bits base on the SRC-DEST table entries. For this example, a naive routing algorithm with limited functionality is to subtract the src-index with the dest-index. Other routing algorithms have been proposed [7], [8].
- *Step vii)* Finally, the streaming values are broadcasted to all Flex-DPEs within a Flex-DPU from the routing bits calculated in Step vi. For reduction, the accumulation ID is processed and then used as the vecID in FAN, described in Section IV-A2. Multicasts, multiplications, and reductions are all happening in a pipelined fashion. Once all the columns have been streamed in and outputs are generated, the Flex-DPE units are freed up to be utilized for another GEMM operation.

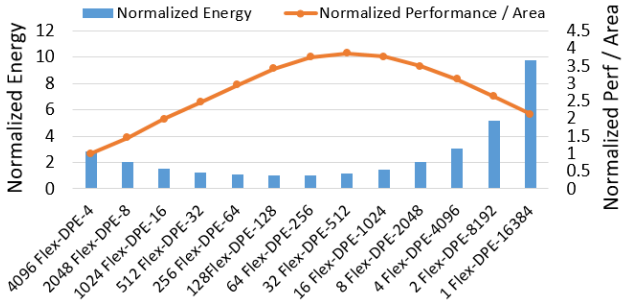


Figure 9: **Design space exploration for Flex-DPE dimensions (256 Flex-DPE-64 is 256 Flex-DPEs of size 64), with aggregated energy consumption (all workloads) and Performance/Area as metrics.**

V. Implementation

Fig. 8 compares the post place-and-routed area and power of a 128×128 systolic array versus SIGMA with 128 Flex-

Metric	Description
Loading Latency	Load Stationary Matrix. Not overlapped with compute.
Streaming Latency	Stream non-stationary Matrix through Distribution Network. Overlaps with Partial-sum generation and Accumulation.
Add Latency	Last reduction before next stationary matrix loaded. Not overlapped with compute.
Total Latency	Loading + Streaming + Add Latency
Stat. Utilization	Percent non-zeros after Stationary Matrix mapped.
Compute Efficiency	Useful (non-zero) MAC Latency / Streaming Latency
Overall Efficiency	Useful (non-zero) MAC Latency / Total Latency

Table II: **Comparison Metrics.**

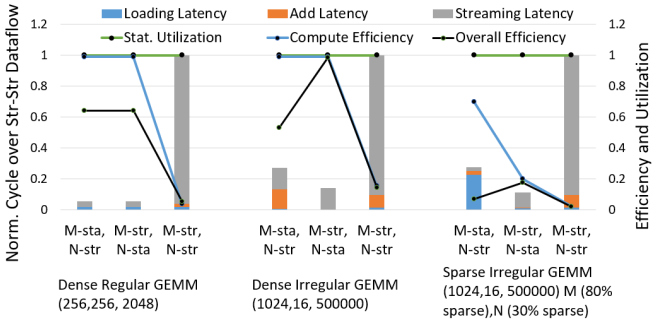


Figure 10: **Comparison of different SIGMA dataflows (described in Fig. 4). with regards to cycle breakdown, utilization, and efficiency. Table II describes the legend.**

DPEs, each of size 128. Both designs have identical input bandwidth of 128 words per cycle from SRAM. SIGMA's key overheads are the highly flexible, non-blocking distribution and reduction networks that lead to a 37.7% area overhead. However, the performance speedups provided by SIGMA (shown later in Sec. VI-C) lead to an average 3.2× improvement in Effective TFLOPs/ Watt. We expect a further power performance gain of 7× when scaling from a 28nm design to a 12nm design. This is based on FP32 FLOPs growth between NVIDIA K20X to NVIDIA T4 where compute grew by ∼2× while power reduced by ∼3.5×. SIGMA is pipelined at 1-cycle distribution, 1-cycle multiplication, and 1-cycle for each level of reduction. The critical path for SIGMA is the distribution, but it is possible to match the maximum operating frequency of TPU by pipelining the distribution further so that the new critical path becomes the FP compute. Additionally, we estimate a global controller with 1024 AND gates, 1024 OR gates, 1024 counters, and 128 SRC-DEST tables to consume approximately $1.4mm^2$.

For 16384 total PEs, we performed a design-space exploration for sizing Flex-DPE units to find the most energy and area efficient configuration. Fig. 9 depicts that a Flex-DPE of size 128 Flex-DPE consumes the least energy, while a Flex-DPE size of 512 is the most area efficient. We decide to use Flex-DPE-128 to match the per-cycle SRAM read bandwidth of the TPU.

VI. Evaluation

A. Methodology

Target GEMMs. We use GEMM dimensions and sparsity observed during training of GNMT, Transformer, NCF and

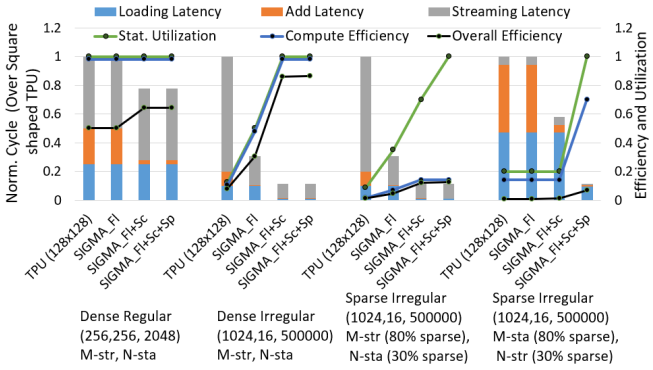


Figure 11: **Performance comparison between TPU and progressive features added in SIGMA. Table II describes the legend. SIGMA_FI adds flexibility over a TPU structure by allowing irregular dimensions. (128x128, 256x64, 512x32 and vice versa) SIGMA_FI+Sc adds scalability with our proposed reduction network FAN. SIGMA_FI+Sc+Sp adds sparsity support to increase utilization.**

DeepBench models (described earlier in Sec. II). Input and weight sparsity were observed to be ∼10-50% and ∼80%.

Baseline Accelerators. We compare SIGMA against other state-of-the-art accelerators: TPU [4], EIE [19], SCNN [33], OuterSPACE [32], Eyeriss v2 [11], Packed Systolic [26], and Cambricon-X [47]. We scale the number of PEs to a constant number of 16384 in all designs. SIGMA assumes 128 Flex-DPEs, each with 128 MACs, and input SRAM bandwidth of 128x (number of unique data elements that can be distributed). For our evaluations, we allow greater input bandwidth to distribute larger chunks of the streaming matrix in one cycle. For sparsity performance, all combinations of matrices and sparsity were tested and then averaged. Most of the sparse accelerators were designed for inference and specialized for convolutions; we extended them to run GEMMs by setting equal input and filter dimensions.

Simulation Infrastructure. To evaluate the performance of SIGMA and other baselines, we developed a cycle-accurate analytic model. The model evaluates the performance based on the number of compute units, buffers per compute unit, interconnect topology, input bandwidth, and dataflow. The TPU was modeled using SCALE-sim [37].

Comparison Metrics. Table II defines comparison metrics we use across our evaluation graphs.

B. Characterizing SIGMA's Features

We start by characterizing the benefits of the various features of SIGMA to help pick the optimal design-point.

Dataflow Comparison Fig. 10 demonstrates the impact of dataflows when running a suite of representative GEMMs. We observe that the MK-str,KN-str dataflow, while being ideal in terms of no wasted computations, suffers in overall latency. This is because it requires extremely high bandwidth (thus serialization), due to no reuse within the Flex-DPE. For the other two dataflows, the stationary matrix maps only non-zeros, getting 100% utilization, and the overall efficiency

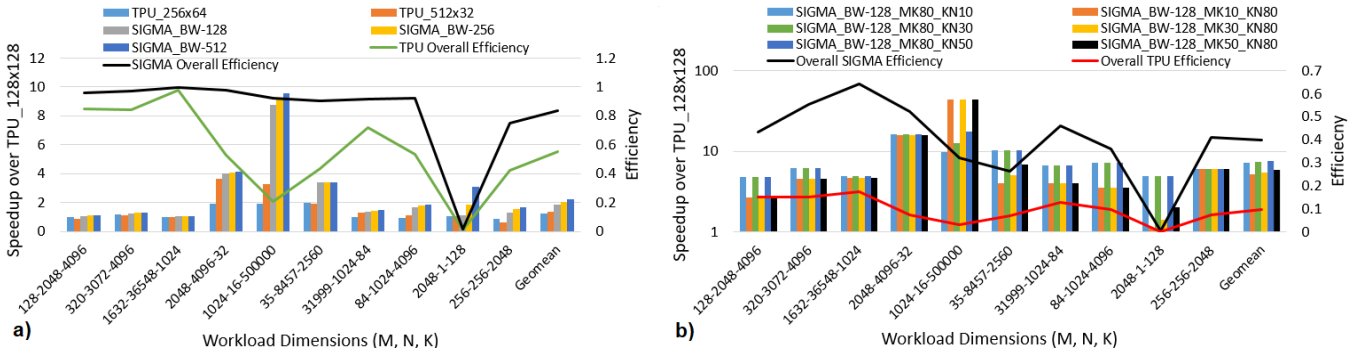


Figure 12: **Speedup and efficiency of (a) rectangular systolic and SIGMA configs over TPU_128×128 for dense workloads, and (b) SIGMA over TPU_128×128 with different input and weight sparsity (MK80 means 80% sparsity in the MK matrix, etc).**

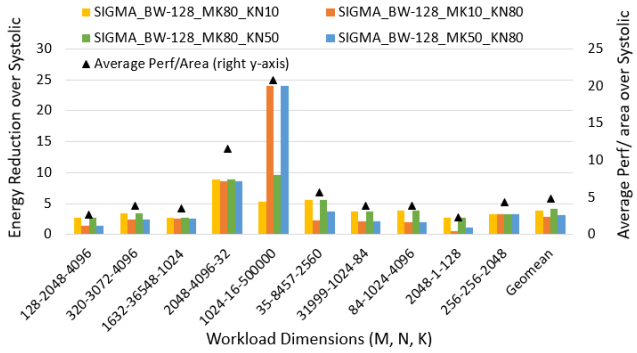


Figure 13: **SIGMA energy reduction over TPU and average performance/ area over TPU's compute array.**

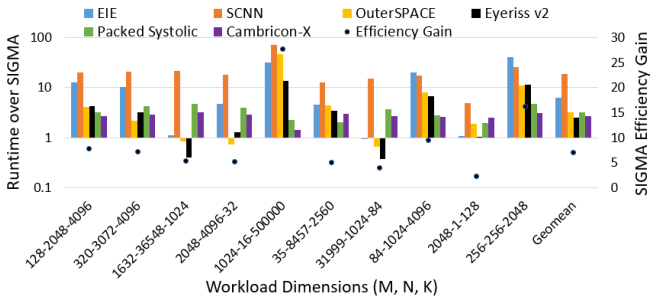


Figure 14: **Comparison between SIGMA and other sparse accelerators with 80% and 30% sparsity on the two matrices.**

gets determined by the sparsity of the streaming matrix. In our evaluations, we run both dataflows and report the higher performing dataflow.

Benefits of SIGMA's features Fig. 11 revisits the discussion from Sec. III-C and quantitatively demonstrates the benefits of SIGMA's three key features in comparison to systolic arrays: (i) flexible dimensions - enabled via FAN for variable sized dot-products within Flex-DPEs, (ii) scalable interconnects, namely, Benes and FAN, providing O(1) and O(log_2N) distribution and reduction time respectively, and (iii) sparsity support to map only useful non-zero data.

For dense regular GEMMs (most left in Fig. 11), SIGMA provides speedup over TPU from its O(1) distribution and O(log_2N) reduction. (TPU has O($sqrt$N) distribution and reduction.) SIGMA's networks reduce Add latency (defined in Table II). The number of cycles saved accumulates whenever the stationary matrix needs to be replaced.

For dense irregular GEMMs, TPU is underutilized if a side of the stationary matrix is smaller than the MAC array dimension. In the example where a 16-500000 sized matrix is stationary; because of a small dimension size of 16, a 128×128 TPU will have a utilization of 12.5%. SIGMA enables full utilization by using rich interconnects that cluster variable dimensions together. Since more PEs are utilized, fewer stationary matrix load iterations are required. This leads to lower loading, add, and streaming latencies.

For sparse irregular GEMMs, TPU is required to map all elements stationary, while SIGMA maps only the nonzeros stationary. With sparsity support, SIGMA shows 100% stationary utilization. Due to increased utilization and compute efficiency, fewer cycles are needed to load and reduce data. Fig. 11 shows two versions of sparse irregular GEMMs. The M-str,N-sta example is dominated by streaming latency because the larger matrix is being streamed in, while the loading latency dominates in M-sta,N-str because the larger matrix is held stationary and leads to more folding iterations. The compute efficiency for M-sta,N-str is significantly higher because the sparser matrix is held stationary.

C. Performance and Energy versus TPU

Speedup. Fig. 12a and Fig. 12b evaluate dense and sparse GEMMs performance respectively. In Fig. 12a, we use three aspect ratios for the TPU. For e.g., 512×32 have 512 rows, each of which can read a data element per cycle. Either the MK or KN matrix is kept stationary. For the 2048-4096-32 GEMM, a K dimension of 32 leads to under-utilization in the 128×128 and 256×64 TPUs, but aligns with the columns of the 512×32 TPU, giving a huge performance jump. SIGMA, due to its flexibility, experiences a similar jump. The TPU overall efficiency drops steeply while operating a 1024-16-500000 sized GEMM. If a square-shaped TPU maps the KN (500000-16) matrix stationary, the low value of N leads to a 87.5% decrease in utilization. If it decides to map

Accelerator	Limitation	SIGMA Solution
TPU [23]	Low utilization from no sparsity support and rigid structure	Flexible interconnects to map non-zero data and irregular GEMMs.
EIE [19]	Not scalable due to all-to-all PE broadcasts and a BW link of one element per cycle	Partition compute to Flex-DPEs (small all-to -all networks) connected by a high BW bus
SCNN [33]	Requires partitioning to use Cartesian product on GEMMs. High inter-PE communications for accumulating outputs.	Multicast GEMM partial sums close to each other so they can be reduced spatially
OuterSPACE [32]	Partial sum accum. within linked list has at best O($NlogN$) complexity	Spatial accum. with our reduction network has O(log_2N) complexity
Eyeriss v2 [11]	Limited weight dist. flexibility and linear reduction	More flexibility with shared all-to-all network and spatial accumulation with novel reduction network FAN.
Packed Systolic [26]	Need algorithmic adjustments and still contains stationary zeros	Bitmap to ensure no zero-valued elements are stationary and no algorithmic changes required.
Cambricon-X [47]	Basic adder tree limits multiplier utilization, allows one common partial sum at a time	FAN enables full multiplier utilization by allowing different partial sums to be accumulated separately.

Table III: **Qualitative Comparision of SIGMA against state-of-the-art accelerators.**

MK (1024-500000) stationary, numerous folds are required since there are only 16K PEs, leading to a large amount of O(N) reductions. SIGMA accelerates this GEMM by creating flexible dimensions and leveraging its O(logN) reduction structure as described in Table I. In SIGMA, the overall efficiency is close to 100% throughout, except for small GEMMs (such as 2048-1-128), where smaller sizes cause loading latency from limited bandwidth to dominate. On average, SIGMA provides speedup of 2x over TPUs on dense GEMMs, stemming from higher utilization and faster reduction. This results to an overall average efficiency of 82% compared to 59% in the TPU. In Fig. 12b we run GEMMs with varying sparsity over SIGMA. We observe that there is a roughly 6× improvement over TPU, which suffers from an average overall efficiency of less than 10% due to the mapped zeros. SIGMA maps no zeros and shows an average overall efficiency of 40%, which gets limited by the sparsity of the streaming matrix.

Energy. In Fig. 13 we see that SIGMA is on an average 3× more energy efficient and 5× more area efficient than TPU for sparse workloads. Despite SIGMA consuming twice as much power (Fig. 8), the energy benefit comes from ∼6× speedup. With more sparsity induced in future workloads, we expect energy gains to be significantly more.

D. Performance against Sparse Accelerators

Fig. 14 presents the speedup of SIGMA over state-of-the-art sparse accelerators. The key inefficiencies in other accelerators are presented in Table III. Of all the sparse accelerators, SIGMA is the only one that can support full spatial-reduction with arbitrary sized dot-products. For two GEMMs, we find SIGMA slower than Eyeriss_v2 since the latter can buffer both operands in its local SRAM for further reuse, while SIGMA keeps only one operand stationary, and has to stream the other multiple times (even if it will be reused in future). Other designs like EIE also have local SRAM buffers, but we observe that its inter-PE communication bottleneck overshadows the memory benefits. On average, we observe SIGMA performing 3X faster than the other sparse accelerators. We tested four combinations between the matrices and sparsity level and selected the best performing one for each accelerator.

VII. Related Work

Training. A few prior works address training on dense matrices. Pipelayer [40] and Neurocube [25] proposes ReRAM based acceleration, but does not address scalability and sparsity. Hypar [41] addresses the scaling problem in training and proposes optimal techniques to extract parallelism. Schuiki et al. [38] and Liu et al. [30] propose processing in memory approaches to combat the communication problem when scaling training. ScaleDEEP architecture was developed to target DNN training, and consists of many processing tiles with specialized memory subsystem and interconnect [44]. However none of these methods simultaneously address the irregularity, sparsity, and scalability as SIGMA does.

Sparsity. Table III contrasts SIGMA against state-of-the-art sparse accelerators. Other recent designs include PermDNN [13], which uses permuted diagonal matrices for inference on structured sparse DNN models. Other designs like UCNN [21] exploits sparsity and weight repetition by reusing dot products. ExTensor [22] finds intersections within compressed representations, and only operates on useful dense computations. Bit-tactical [12] targets sparsity in inference by skipping zero weights and exploiting bit level sparsity of inputs. Unlike SIGMA, Bit-tactical leverages scheduling in software to align inputs and weights. SparseReRAM [46] proposes using small operation units to exploit both weight and activation sparsity in ReRAMs. SIGMA targets acceleration of GEMMs with unstructured sparsity.

Flexibile Interconnects. Eyeriss [10] proposes an efficient dataflow for leveraging convolutional reuse with reconfigurable buses. MAERI [27] uses tree-based interconnects for distribution and reduction which inspired the 1D Flex-DPE microarchitecture in SIGMA. However, MAERI does not handle dynamic weight and activation sparsity, and is optimized for low-precision CNNs rather than high-precision GEMMs commonly used during DNN training. Eyeriss v2 [11] also uses a specialized NoC to handle sparsity, but is optimized for small mobile CNNs rather than large GEMMs.

VIII. Conclusion

The paper proposes SIGMA as an accelerator to handle emerging large, sparse, and irregular GEMMs. SIGMA provides close to 100% compute utilization via high-bandwidth non-blocking interconnect structures. The overhead for such flexibility is carefully analyzed via a place-and-routed design. Our implementation shows 5.7× performance speedup over TPU designs for sparse irregular workloads. We also observe a 3× performance speedup over other state-of-the-art sparse accelerators. Reference RTL:

https://github.com/georgia-tech-synergy-lab/SIGMA

Acknowledgement

We thank Ishwar Bhati and Sasikanth Avancha for helpful technical discussions. This work was funded by Intel.

References

[1] "https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/," 2015.
[2] "Baidu-deep bench," 2016.
[3] "Nvdla deep learning accelerator," in *http://nvdla.org*, 2018.
[4] "Cloud tpu," in *https://cloud.google.com/tpu*, 2019.
[5] "https://docs.nvidia.com/cuda/cusparse/index.html," 2019.
[6] D. Amodei and D. Hernandez, "https://blog.openai.com/ai-and-compute/," 2018.
[7] S. Arora *et al.*, "On-line algorithms for path selection in a nonblocking network," in *STOC*, 1990.
[8] A. Chakrabarty *et al.*, "Matrix-based nonblocking routing algorithm for benes networks," in *Computation World*, 2009.
[9] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.
[10] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *ISSCC*, 2016.
[11] Y. Chen *et al.*, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," in *JETCAS*, 2019.
[12] A. Delmas Lascorz *et al.*, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *ASPLOS*, 2019.
[13] C. Deng *et al.*, "Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices," in *MICRO*, 2018.
[14] J. Fowers *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *ISCA*, 2018.
[15] T. Gale *et al.*, "The state of sparsity in deep neural networks," *arXiv:1902.09574v1 [cs.LG]*, 2019.
[16] M. Gao *et al.*, "Tangram: Optimized coarse-grained dataflow for scalable nn accelerators," in *ASPLOS*, 2019.
[17] A. Gondimalla *et al.*, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *MICRO*, 2019.
[18] S. Hadjis *et al.*, "Caffe con troll: Shallow ideas to speed up deep learning," in *DanaC*, 2015.
[19] S. Han *et al.*, "Eie: efficient inference engine on compressed deep neural network," in *ISCA*, 2016.
[20] X. He *et al.*, "Neural collaborative filtering," 2017.
[21] K. Hedge *et al.*, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *ISCA*, 2018.
[22] K. Hegde *et al.*, "Extensor: An accelerator for sparse tensor algebra," in *MICRO*, 2019.
[24] K. Kanellopoulos *et al.*, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *MICRO*, 2019.
[23] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.
[25] D. Kim *et al.*, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *ISCA*, 2016.
[26] H. Kung *et al.*, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *ASPLOS*, 2019.
[27] H. Kwon, *et al.*, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *ASPLOS*, 2018.
[28] H. Kwon *et al.*, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in *MICRO*, 2019.
[29] T. T. Lee and S. Y. Liew, "Parallel routing algorithms in benes–clos networks," in *IEEE Trans. Commun.*, 2002.
[30] J. Liu *et al.*, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *MICRO*, 2018.
[31] Nvidia, "Nvidia tesla v100 gpu architecture," in *Volta Architecture Whitepaper*, 2017.
[32] S. Pal *et al.*, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *ISCA*, 2018.
[33] A. Parashar *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.
[34] J. Park *et al.*, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," *CoRR*, vol. abs/1811.09886, 2018.
[35] L. Pentecost *et al.*, "Maxnvm: Maximizing dnn storage density and inference efficiency with sparse encoding and error mitigation," in *MICRO*, 2019.
[36] M. Rhu *et al.*, "Compressing dma engine: Leveraging activation sparsity for training deep neural networks," in *HPCA*, 2018.
[37] A. Samajdar *et al.*, "Scale-sim: Systolic cnn accelerator simulator," in *arXiv:1811.02883v1 [cs.DC] 16 Oct 2018*.
[38] F. Schuiki *et al.*, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *IEEE Transactions on Computers*, 2019.
[39] Y. S. Shao *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *MICRO*, 2019.
[40] L. Song, *et al.*, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *HPCA*, 2017.
[41] L. Song *et al.*, "Hypar: Towards hybrid parallelism for deep learning accelerator array," *HPCA*, 2019.
[42] A. Vaswani, *et al.*, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017.
[43] A. Vaswani *et al.*, "Tensor2tensor for neural machine translation," *CoRR*, vol. abs/1803.07416, 2018.
[44] S. Venkataramani *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *ISCA*, 2017.
[45] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016.
[46] T.-H. Yang *et al.*, "Sparse reram engine: joint exploration of activation and weight sparsity in compressed neural networks," in *ISCA*, 2019.
[47] S. Zhang *et al.*, "Cambricon-x: An accelerator for sparse neural networks," in *MICRO*, 2016.
[48] M. H. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv:1710.01878v2 [stat.ML]*, 2017.