



Systolic Sparse Tensor Slices: FPGA Building Blocks for Sparse and Dense AI Acceleration

Endri Taka
The University of Texas at Austin
Austin, TX, United States
endri.taka@utexas.edu

Ning-Chi Huang
National Yang Ming Chiao Tung
University
Hsinchu, Taiwan

Chi-Chih Chang
National Yang Ming Chiao Tung
University
Hsinchu, Taiwan

Kai-Chiang Wu
National Yang Ming Chiao Tung
University
Hsinchu, Taiwan

Aman Arora
Arizona State University
Tempe, AZ, United States

Diana Marculescu
The University of Texas at Austin
Austin, TX, United States

Abstract

FPGA architectures have recently been enhanced to meet the substantial computational demands of modern deep neural networks (DNNs). To this end, both FPGA vendors and academic researchers have proposed in-fabric blocks that perform efficient tensor computations. However, these blocks are primarily optimized for dense computation, while most DNNs exhibit sparsity. To address this limitation, we propose incorporating *structured* sparsity support into FPGA architectures. We architect 2D systolic in-fabric blocks, named systolic sparse tensor (SST) slices, that support multiple degrees of sparsity to efficiently accelerate a wide variety of DNNs. SSTs support dense operation, 2:4 (50%) and 1:4 (75%) sparsity, as well as a new 1:3 (66.7%) sparsity level to further increase flexibility. When demonstrating on general matrix multiplication (GEMM) accelerators, which are the heart of most current DNN accelerators, our sparse SST-based designs attain up to 5× higher FPGA frequency and 10.9× lower area, compared to traditional FPGAs. Moreover, evaluation of the proposed SSTs on state-of-the-art sparse ViT and CNN models exhibits up to 3.52× speedup with minimal area increase of up to 13.3%, compared to dense in-fabric acceleration.

CCS Concepts

• **Hardware** → **Hardware accelerators**; • **Computer systems organization** → **Systolic arrays**.

Keywords

FPGA, structured sparsity, hardware acceleration, matrix multiplication, computer architecture, deep learning, machine learning

ACM Reference Format:

Endri Taka, Ning-Chi Huang, Chi-Chih Chang, Kai-Chiang Wu, Aman Arora, and Diana Marculescu. 2025. Systolic Sparse Tensor Slices: FPGA Building Blocks for Sparse and Dense AI Acceleration. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '25)*, February 27–March 1, 2025, Monterey, CA, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3706628.3708867>



This work is licensed under a Creative Commons Attribution International 4.0 License.

FPGA '25, February 27–March 1, 2025, Monterey, CA, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1396-5/25/02
<https://doi.org/10.1145/3706628.3708867>

1 Introduction

Sparsity, which refers to zeros in weight or activation tensors, is an inherent attribute of contemporary deep neural networks (DNNs) [35]. Sparsity arises from complex interactions among various optimization techniques in modern DNN models [81]. For instance, over-parameterization is prevalent because it enables easier training and better generalization [35]. Pruning is typically applied for this over-parameterization redundancy, resulting in zeros within the weight tensors (sparsity). The exploitation of sparsity to reduce computational and memory requirements has been a common strategy in the design of efficient DNN accelerators [57].

Typically, zeros in DNNs are distributed randomly within the data tensors [35, 57]. This random sparsity is commonly referred to as *unstructured* sparsity. Leveraging unstructured sparsity has been the main focus of numerous sparse hardware accelerators over the past years, targeting both ASICs [21, 29, 32, 34, 65, 68, 87, 89] as well as FPGAs [37, 45, 52, 58, 59, 82]. However, unstructured sparse accelerators require complex hardware structures, which result in high area overheads [81]. This area increase leads to substantial rise in energy consumption, *e.g.*, up to 71% [57], compared to dense architectures. Moreover, the random location of zeros in unstructured sparsity causes unpredictable and low hardware utilization, rendering inference speedup inefficient as well as challenging [32, 55, 56].

To address these challenges in sparsity acceleration, *structured* sparsity has been proposed recently [47, 57, 60]. Structured sparsity imposes constraints on the sparsity patterns in data tensors, enabling low area overheads and highly energy-efficient hardware enhancements for sparsity exploitation. To this end, Nvidia A100 GPUs introduced support for the 2:4 structured sparsity, where two out of every four *consecutive* values must be non-zero. Furthermore, structured sparse ASIC accelerators have also been proposed, demonstrating up to 3.1× less energy consumption per inference compared to state-of-the-art unstructured sparse accelerators [57].

Alongside GPUs and ASICs, FPGAs have emerged as a promising candidate for accelerating the rapidly evolving DNNs, due to their high flexibility of reconfiguration. Contemporary FPGA architectures have been enhanced to more effectively support the high computational demands of DNNs. In particular, Intel employs in-fabric blocks comprising multiple dot-product engines [30, 51], while AMD introduced out-of-fabric programmable vector processors [11]. Academic researchers have also proposed in-fabric blocks

that employ a 2D systolic dataflow [13, 14], exhibiting substantial benefits over traditional FPGAs for various DNN workloads.

However, these in-fabric blocks are primarily designed for dense DNN acceleration, limiting their efficiency and applicability to most real-world DNNs, which inherently exhibit sparsity. To address this challenge, we propose incorporating flexible in-fabric slices into the FPGA architecture to efficiently support both *sparse* and *dense* DNN workloads. We employ 2D systolic slices similar to [13, 14], which are augmented with structured sparsity features and further optimizations. Our novel systolic sparse tensor (SST) slices are architected with the following principal properties:

- **Efficiency.** (i) Sparsity features should exhibit low area overheads. (ii) Translation of sparsity to actual DNN acceleration. (iii) Maximization of data reuse in sparse and dense DNNs. (iv) Efficient sparse format for storing the non-zero values.
- **Sparsity Flexibility.** These in-fabric blocks must be flexible, *i.e.*, support multiple structured sparsity levels (percentage of zeros) to accelerate a wide range of DNN workloads.

The SST slices are systolic-based and utilize a highly efficient index-based sparse format, effectively meeting the aforementioned key properties for *efficiency*. Regarding the *sparsity flexibility*, SSTs support multiple sparsity levels, *i.e.*, dense, 2:4 (50% sparsity), 1:3 (66.7%) and 1:4 (75%). These levels align with the most *common* sparsity levels in DNNs, since sparsity higher than 75% typically leads to significant accuracy degradation [27, 55, 88, 90, 91]. Prior works have exploited the 2:4 and 1:4 sparsity patterns to design sparse accelerators using traditional FPGAs [27, 83], while others have incorporated these patterns into CPU datapaths [47, 73]. In contrast, we propose a novel in-fabric FPGA block that additionally supports a new 1:3 sparsity pattern. The 1:3 sparsity bridges the gap between 50% and 75% sparsity, increasing flexibility and providing both acceleration and storage benefits. Furthermore, 1:3 sparsity can be supported in SSTs *without incurring additional area overheads*, by effectively reusing the area allocated for 2:4 and 1:4. This sparsity flexibility enables the development of tailored solutions for DNN models, since each model exhibits distinct levels of sparsity.

The proposed SST slices support 8-bit integer (int8) and brain floating-point (bfloat16) precisions, since these are the most commonly used data types in DNN accelerators [48, 62]. Additionally, we introduce *dedicated* interconnects between the SST slices to facilitate their efficient integration within the FPGA architecture. These dedicated interconnects demonstrate substantial routing savings in FPGAs compared to prior work on 2D systolic blocks [13, 14], where such interconnects are not employed.

The SST slices deliver highly efficient acceleration of sparse and dense general matrix multiplication (GEMM) in the FPGA fabric. Our main focus is GEMM, as most current state-of-the-art DNNs across various applications are Transformer-based [31, 44], with GEMM serving as the core computation. To the best of our knowledge, this is the first work to support structured sparsity in the FPGA architecture. Our key contributions are summarized below:

- A *generalizable methodology* for incorporating structured sparsity into in-fabric FPGA blocks. Our proposed SST slices employ a 2D systolic dataflow and support multiple levels of sparsity. Besides dense, 2:4 and 1:4, we also introduce a *new 1:3 sparsity pattern* to provide greater flexibility, enabling efficient acceleration for the majority of DNN models.

- We introduce dedicated interconnects to effectively integrate the SST slices in the FPGA architecture. These interconnects show *significant routing savings, up to 31.2%* on GEMM accelerators, compared to utilizing only global routing resources, as proposed in prior work on 2D systolic blocks.
- Demonstration on sparse GEMM accelerators leveraging our SST slices show considerably *higher attainable FPGA frequency, up to 4.4× for int8 and 5× for bfloat16*, as well as *remarkable area reduction, up to 7× for int8 and 10.9× for bfloat16*, when comparing with traditional FPGAs.
- Our evaluation on state-of-the-art sparse ViT and CNN models showcases up to **3.52× speedup** when exploiting our SST slices compared to dense in-fabric blocks, with *low area overheads of 10.2% and 13.3% for int8 and bfloat16*, respectively.

2 Related Work

Structured sparsity is currently supported by various commercial architectures. In particular, the 2:4 sparsity is supported by state-of-the-art GPUs, *i.e.*, Nvidia A100 [63], Nvidia H100 [64] and AMD MI300 [7], as well as the new AMD Versal AIE-ML processors [8, 11]. In academic research, hardware support for structured sparsity has also been investigated in ASIC DNN accelerators [56, 57, 81]. Moreover, prior academic research [91] introduces structured sparsity features in the tensor cores of modern GPUs. Finally, structured sparsity support has also been proposed inside the matrix engines [47] and vector units [73] of CPUs. Considering all prior work, we are the first to incorporate structured sparsity support into the FPGA architecture. Moreover, our proposed in-fabric blocks support multiple levels of structured sparsity (beyond merely 2:4), to enable efficient acceleration for most contemporary DNN models.

In-fabric hard blocks have been commercially incorporated into several AI-optimized FPGAs. More specifically, the Intel Stratix 10 NX [51] replaces the traditional DSP slices with tensor blocks. These tensor blocks comprise multiple dot-product engines to more efficiently support AI applications. Very recently, Intel announced the Agilex-5 FPGAs [30, 41], which introduce new AI-enhanced DSP blocks. These new DSPs include multiple dot-product operations, while also maintaining various features of the traditional Agilex DSP blocks. Finally, the Achronix Speedster7t FPGAs incorporate machine learning processor (MLP) blocks [2, 19]. These MLP hard blocks feature multiplier arrays, adder trees, accumulators and tightly coupled memories to the computational blocks.

In-fabric enhancements have also been proposed in academia. In [13, 14], the authors enrich the FPGA architecture with 2D systolic tensor slices, showing substantial efficiency benefits in DNN acceleration over traditional FPGAs. Moreover, in [66], a special pattern of dedicated wires between DSP blocks is proposed, which enables more efficient mapping of systolic arrays on FPGAs. However, all aforementioned commercial and academic in-fabric blocks target primarily dense computation. In this work, we architect in-fabric slices that enable both sparse and dense computation. We propose incorporating the SST slices, which employ a 2D systolic dataflow similar to [13, 14], but are augmented with sparsity features. Moreover, we introduce vertical dedicated wires between the SSTs to more efficiently map GEMM accelerators into FPGA architectures.

Finally, prior research [18, 20, 27, 53, 83] implement DNN accelerators that support multi-level structured sparsity on traditional

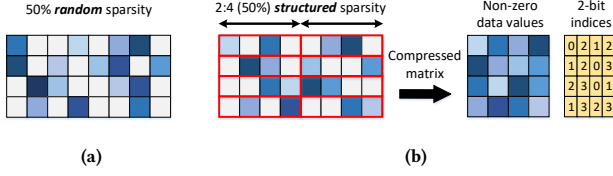


Figure 1: 50% unstructured sparse matrix (a) and 2:4 (50%) structured sparse matrix along with compressed format (b).

(non AI-optimized) FPGAs. In contrast, we also introduce the novel 1:3 sparsity pattern, which is absent from any prior work and offers increased sparsity flexibility. In this work, we demonstrate the importance of this flexibility for various state-of-the-art DNNs. Furthermore, our SST-based GEMM designs show remarkable performance and area efficiency gains compared to traditional FPGAs.

3 Architecture & Design Overview

3.1 Fine-Grained Structured Sparsity

In this work, we leverage the regular patterns of fine-grained structured sparsity to architect SST slices with low area overhead. Fig. 1a depicts a 50% unstructured sparse matrix, where the non-zero data are distributed *randomly*, *i.e.*, there is no specific pattern of their locations. In contrast, in Fig. 1b, the 2:4 structured sparsity pattern is illustrated, which has the same sparsity level (50%), but in every group of four *consecutive* elements there are two non-zero values. Notice that the location of the two non-zero values can vary significantly within the four-element group, offering *fine-grained* sparsity flexibility. These types of constraints enable low area hardware enhancements, allowing for efficient exploitation of sparsity.

A 2:4 sparse matrix can be efficiently stored in compressed format by saving only the non-zero values. The location of each non-zero data is encoded using 2-bit indices, as shown in Fig. 1b. Similar to 2:4, for 1:4 (75%) sparsity one every four consecutive elements is non-zero, while for 1:3 (66.7%) sparsity there is one non-zero every three consecutive elements. For all aforementioned patterns, 2-bit indices are required to encode the location of each non-zero element. This is a very efficient compressed format, as we show in Sec. 4.6, where comparison among other formats is performed.

3.2 Systolic Sparse Tensor Slices Architecture

In this section, we present an overview of the proposed SST slices. The core compute unit of the SSTs is a 4×4 systolic array (SA) [50], as depicted in Fig. 2. A 2D SA consists of homogeneous processing elements (PEs), where each PE performs a multiply-accumulate (MAC) operation and forwards the input operands to the neighboring PEs. This architecture allows maximization of data reuse in GEMM, while also delivering high performance due to its regular and highly scalable design. Hence, SAs have become a prime architecture in many DNN accelerators [47, 48, 57, 62, 67, 80, 85]. In this work, we show that incorporating SST slices in FPGAs leads to high performance and scalable dense/sparse GEMM accelerators.

3.2.1 SST Operation. We enhance the systolic PEs with sparse features by introducing sparse processing elements (SPEs), while maintaining the properties of the SAs discussed above. Our SSTs utilize an output stationary SA, consisting of 16 SPEs. The accumulations remain stationary in the SPEs, while input operands are propagated to their neighbors every clock cycle. The a_data of an

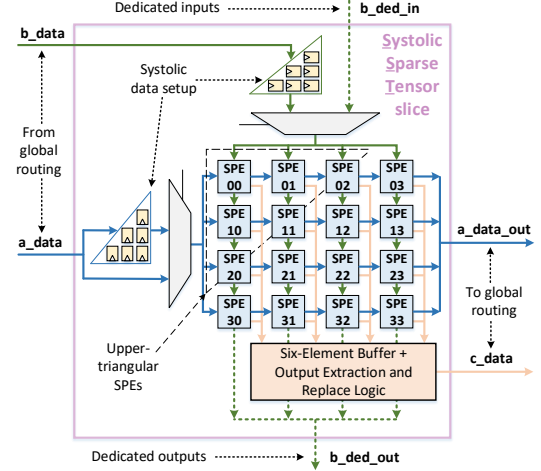


Figure 2: Systolic Sparse Tensor slice architecture.

input matrix A are propagated and reused across SPEs horizontally, while the b_data of an input matrix B are propagated and reused vertically (Fig. 2). The matrix A can be either sparse or dense (typically to map *weights*), while B is dense (typically to map input *activations*). The architecture of the SPEs is delineated in Sec. 3.3.

Besides the 4×4 SPE grid, we also implement pipeline registers to delay the input operands for systolic data setup [49] (arranged in triangular manner in Fig. 2). Systolic setup is needed at the interface for loading input matrices (typically from on-chip memory), when chaining multiple SSTs to construct larger SA grids (Sec. 3.4). Multiplexers are used to select either the systolic setup or directly the input data, and are configured *statically* (during bitstream loading).

The SSTs support both int8 and bfloat16 precisions, while accumulations are realized in 32-bit integer (int32) and IEEE 32-bit floating-point (fp32), respectively, similar to Nvidia GPUs [60] and Google TPUs [62]. When the SA operation is completed, the output matrix C is extracted via the c_data output ports. We note that SPEs finish their operation in a *diagonal* fashion cycle after cycle. In the first cycle, $SPE00$ finishes, in the second cycle both $SPE01$ and $SPE10$ finish, etc. (Fig. 2). However, to maintain *regularity*, thus simplifying the downstream logic (typically in CLBs), we extract the output values in a column-wise manner (four values per cycle). To achieve that, we introduce a six-element buffer (consisting of registers), to store the data before getting extracted. This is particularly important in sustaining 100% SPE utilization (16 MACs per cycle) at the steady-state (matrices processed one after the other).

Since SPEs complete their operation diagonally, the six upper-triangular SPE outputs, shown in Fig. 2, need to be stored in the buffer. Suppose a cycle T where the outputs of the biggest diagonal, *i.e.*, $SPEs$ {03, 12, 21, 30} are generated. In cycle T , the values of the first column, *i.e.*, $SPEs$ {00, 10, 20, 30} can be extracted, where $SPEs$ {00, 10, 20} are loaded from the buffer, while only $SPE30$ is directly extracted. In cycle $T + 1$, the values of $SPEs$ {03, 12, 21} replace the position of $SPEs$ {00, 10, 20} in the buffer (since they have been extracted). Therefore, the locations of the six-element buffer are being effectively reused, and in cycle $T + 1$, the second column can be extracted, *i.e.*, $SPEs$ {01, 11, 21, 31}. In a similar fashion, the rest two columns are extracted in cycles $T + 2$ and $T + 3$, respectively, while the buffer locations are efficiently reused due to replacement.

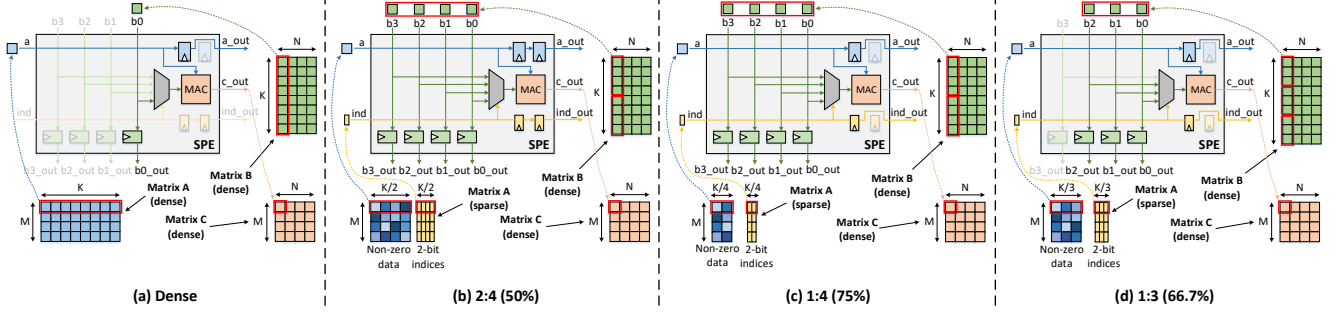


Figure 3: Sparsity modes in Systolic Sparse Elements of the SST slices (multiplexing logic omitted for clarity).

3.2.2 Global Routing Interface & Dedicated Wires. As illustrated in Fig. 2, the a_data and b_data input ports as well as c_data output ports are connected to the global FPGA routing resources. When chaining multiple SSTs, a_data and b_data are forwarded to their next in the chain SSTs, horizontally and vertically, respectively. Horizontally, the data are propagated via the a_data_out using the FPGA routing resources. However, vertically, we utilize *dedicated* wires to propagate the data (via the b_ded_out ports) and connect them to the next SST in the *same* FPGA column (via the b_ded_in ports). These vertical dedicated wires provide efficient connections without the usage of the global routing resources, matching the columnar nature of the modern FPGA fabric [15, 16]. This approach significantly reduces routing resources, as opposed to [13, 14], where all inputs/outputs (I/Os) of the in-fabric tensor slices are connected to global routing (see comparison in Sec. 4.3).

Besides the I/Os shown in Fig. 2, the SSTs include also several dynamic control signals. In particular, an input *enable* signal is used to control the operation of the SSTs. This signal should be deasserted when the operation of the SST needs to be stalled. Moreover, an *accumulate* signal is utilized to control the accumulation in the SPEs. This signal remains asserted when accumulation is desirable in the SPEs, e.g., during tiling to process larger matrices. However, for every new matrix operation the *accumulate* signal should be deasserted for one cycle. Another input signal is the d_type , which dynamically selects the precision, i.e., int8 or bfloat16. Finally, a 2-bit *sparsity_level* signal is employed to dynamically select the sparsity level of the matrix A, i.e., dense, 2:4, 1:3, 1:4. This signal is utilized internally in the SSTs to control the operation of the supported sparsity modes, as discussed in the next section.

Regarding the outputs of the SST, an *accumulate_out* signal is used for systolic distribution of the *accumulate* signal when chaining multiple SSTs. This systolic distribution makes the control logic significantly simpler, since the *accumulate* is set only for the first SST in the 2D array layout, while being distributed to the remaining SSTs, as explained in Sec. 3.4. Another output signal is the *valid_out*, which is asserted when the output c_data are extracted column-wise (Fig. 2). This signal simplifies the downstream logic, since only *valid_out* needs to be checked for valid output data.

3.3 Sparse Processing Element

The SPE architecture in all supported sparsity modes is illustrated in Fig. 3. When the SPE is configured in dense mode (Fig. 3a), it operates as a regular dense PE. In particular, one MAC operation is performed every cycle, while the elements of matrices A and

Table 1: Summary of supported sparsity levels in SST slices.

Sparsity level	Compres. ratio		Speedup over dense	SPE util.
	int8	bfloat16		
Dense (0%)	1×	1×	1×	100%
2:4 (50%)	1.6×	1.78×	2×	100%
1:3 (66.7%)	2.4×	2.67×	3×	100%
1:4 (75%)	3.2×	3.56×	4×	100%

B are forwarded to their neighboring SPEs (via pipeline registers) horizontally and vertically, respectively. The output value of matrix C in each SPE is calculated after K cycles, when considering the $M \times K \times N$ matrix dimensions depicted in Fig. 3a.

In 2:4 mode (50% sparsity), matrix A is stored in a compressed format of $M \times K/2$ size for both values and the 2-bit indices, as shown in Fig. 3b. In order to achieve speedup over the dense case, we increase the number of ports of each SPE in the vertical dimension, to load four elements of the matrix B in parallel. Furthermore, the 2-bit index is also loaded in the SPE to select the corresponding B values (via a 4:1 multiplexer), which need to get multiplied with each A value. However, in 2:4 sparsity, for every four values of the matrix B, two MAC operations need to be performed. Hence, the four B values remain in the SPE registers for two clock cycles (the four green registers in Fig. 3b are loaded every two cycles). To correctly synchronize the systolic operation, two pipeline stages are required for both the A values and their indices (blue and yellow registers in Fig. 3b), which are loaded every cycle. In this manner, one MAC operation is performed every cycle, ensuring 100% utilization. Moreover, the output value of matrix C in each SPE is calculated in $K/2$ cycles, achieving 2× acceleration over dense operation.

Similar to 2:4, for 1:4 (75%) sparsity, four B values are loaded in the SPE. However, in this case the B values are loaded every cycle, while only one pipeline stage is required for the A value and its index, as shown in Fig. 3c. The output value needs $K/4$ cycles to be calculated, offering 4× speedup over the dense case. To fill the sparsity gap between 50% and 75% as explained in Sec. 1, we leverage the same hardware enhancements for 2:4 and 1:4 sparsity to also support the 1:3 (66.7%) pattern. In particular, as illustrated in Fig. 3d, the 1:3 operation is similar to 1:4, with the main difference being that three B values are loaded every cycle instead of four. The indices ensure that the fourth B value is never selected, thus no additional hardware is required. In this case, $K/3$ cycles are needed for every SPE output value, providing 3× acceleration. Finally, we note that similar to 1:3, 2:3 sparsity can also be supported by directly utilizing the 2:4 mode, leading to 33.3% sparsity. However, a matrix is typically considered sparse when it has sparsity of 50% or higher [8]. Hence, in this work, we do not consider the 2:3 pattern.

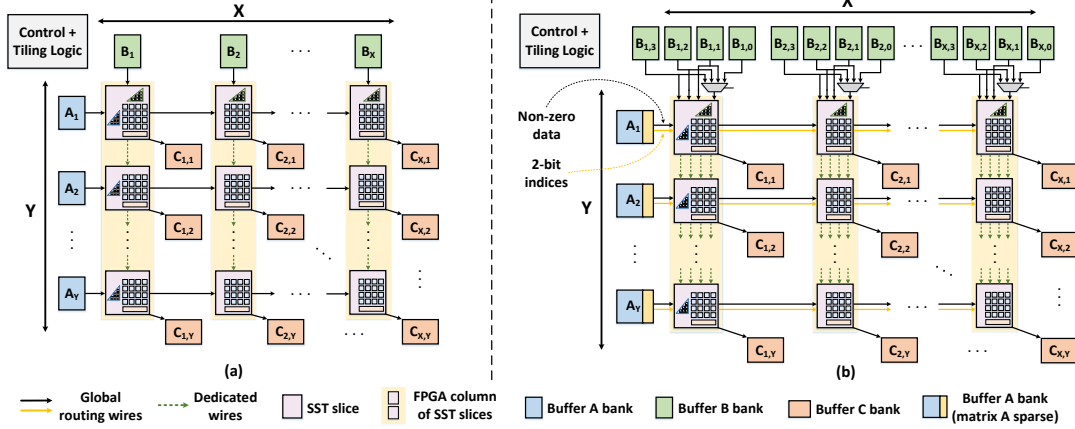


Figure 4: 2D systolic GEMM design: dense implementation (a) and dynamic configuration of all supported sparsity modes (b).

In Table 1, we present a summary of the supported sparsity levels. First, we show the *compression ratio*, *i.e.*, the memory reduction over dense storage due to compressed format, for both int8 and bfloat16 precisions. For all sparsity levels, bfloat16 offers a higher compression ratio over int8, *e.g.*, $1.78\times$ vs. $1.6\times$ for 2:4 sparsity. This is because 2-bit indices are required for both 8-bit and 16-bit data types, resulting in relatively lower overhead for the compressed representation in bfloat16 compared to int8. This compressed format substantially reduces both on-chip and off-chip memory requirements, achieving up to $3.56\times$ reduction (Table 1).

Second, we notice that every sparsity level is translated to its corresponding speedup, *e.g.*, $4\times$ for 1:4 (75%) sparsity, achieving 100% SPE utilization in all cases (Table 1). For all sparsity levels, data reuse is maximized since both indices and values are propagated and reused horizontally and vertically, similar to dense operation.

3.4 GEMM Design Utilizing Multiple SST Slices

In this section, we describe parametric GEMM implementations for both dense and sparse configurations, utilizing multiple SST slices. Both implementations are highly regular and scale effectively on the FPGA fabric, attaining high frequencies as shown in Sec. 4.4.

3.4.1 Dense Implementation. Fig. 4a depicts a parametric GEMM accelerator comprising a 2D array of SST slices, which are configured in *dense* mode (Sec. 3.3). The 2D array consists of $Y \cdot X$ SST slices, implementing a total SA size of $(Y \cdot 4) \times (X \cdot 4)$. This size is denoted as the *native* size of the GEMM accelerator. On-chip memory buffers are implemented to store the input matrices A , B and the output matrix C . The input buffers A , B are located in the left and top edges of the 2D array, respectively, and are partitioned into banks, providing sufficient bandwidth to feed the SSTs. In particular, for buffer A , Y banks are required, while for buffer B , X banks are needed. Since each SST slice includes a 4×4 SA, each bank needs to provide a bandwidth of 32-bits per cycle when SSTs are configured for int8 precision, while for bfloat16, 64-bits per cycle are required. Regarding the output buffer C , $X \cdot Y$ banks are needed due to the output stationary architecture of the SSTs, each receiving an output of 128-bits per cycle (four 32-bit values as explained in Sec. 3.2).

The data from the buffers A , B are propagated in a systolic fashion between the SST slices, as illustrated in Fig. 4a. Horizontally, the A data are propagated via the global routing resources of the FPGA

fabric. Vertically, the B data are inserted via global routing wires in the first SST at each vertical chain (Y SSTs in Fig. 4a), while being forwarded to the next SSTs via dedicated wires. It is important to note here that the SSTs comprising each vertical chain are *physically* contiguous in the FPGA. Nevertheless, in the horizontal dimension, the Y chains might not follow the *logical* arrangement shown in Fig. 4a inside the FPGA fabric, due to the routing flexibility of FPGAs. This depends on decisions made by the FPGA place and route (PnR) algorithm. Finally, notice the (static) systolic data setup configuration specifically for the SST slices that interface with the buffers A , B (left and top edge of the 2D array).

Control logic, mapped to the CLB resources of the FPGA, is utilized to orchestrate the entire operation of the GEMM design. We also implement tiling logic (in CLBs) to exploit data reuse in GEMM, as well as to support arbitrary GEMM sizes based on the available on-chip memory resources. When mapping an arbitrary GEMM of $M' \times K' \times N'$ dimensions, M' must be a multiple of $(Y \cdot 4)$, while N' must be a multiple of $(X \cdot 4)$, since the *native* size of the accelerator is $(Y \cdot 4) \times (X \cdot 4)$. Note that there is no constraint on the reduction K' dimension. Finally, although not shown in Fig. 4a, *accumulate* signals utilized during tiling (Sec. 3.2) are propagated in a systolic fashion among the 2D array of SSTs, similar to the A , B data. The control logic sets the *accumulate* signal only for the first SST in both vertical and horizontal dimensions (*i.e.*, the SST fed by buffers A_1 and B_1), which significantly simplifies the overall logic.

3.4.2 Dynamic Sparse Configuration. Fig. 4b illustrates a parametric GEMM design that is *dynamically* configured to support all the sparsity modes in the SSTs, *i.e.*, dense, 2:4, 1:3 and 1:4. This dynamic configuration is particularly important for layer-wise sparsity exploitation in DNNs, since each layer might require different sparsity level for optimal trade-off between DNN accuracy and speedup (Sec. 4.5). We note that the implementation is similar to the dense design (Fig. 4a), with main differences lying in the design of buffers A and B , as well as in the vertical and horizontal propagation of the data. Horizontally, the A data are kept in the buffer banks in compressed format for the sparse modes (2:4, 1:3 and 1:4). In this case, both non-zero data and indices are loaded in the SSTs and are propagated horizontally (see Sec. 3.3). More specifically, for int8, each buffer A bank needs to provide a bandwidth of 40-bits per cycle, due to the additional 8-bits for the four vertical SPEs at the interface of each SST. Similarly, for bfloat16, 72-bits per cycle are required.

Table 2: FPGA logic tile and routing parameters in COFFE.

Parameter	Value	Parameter	Value
FLEs per CLB (N)	10	Frac. LUT size (K)	6
FLE independent inputs	2	Adders per FLE	2
Channel width (W)	300	Wire length (L)	4/16
CLB inputs (I)	60	CLB outputs (O)	40
FLE outputs to routing (O_r)	4	Feedback FLE outputs (O_{fb})	2
SB flexibility (F_s)	3	Input connection flex. (F_{cin})	0.15
Output connection flex. (F_{cout})	0.1	Input crossbar flex. (F_{local})	0.5

Vertically, four B banks are needed to feed the SSTs due to the $4\times$ increase in ports for 2:4 and 1:4 sparsity acceleration (Sec. 3.3). Each B bank provides the same bandwidth as the dense design (Fig. 4a), *i.e.*, 32-bits and 64-bits per cycle for int8 and bfloat16, respectively. Similar to the dense design, the B data are inserted in the first SST at each vertical chain and dedicated wires are used to propagate them vertically (Fig. 4b). These dedicated wires are particularly important for the sparse design, since otherwise $4\times$ more vertical wires would be required to use global routing compared to the dense design (in the case where *only* non-dedicated wires are employed).

Besides sparse operation, the design in Fig. 4b also supports dense computation. This is because dense computation might still be needed, even if all weights in DNNs are sparse. For instance, in Transformer-based DNNs [25, 26, 77], the QKV (Query, Key, Value) GEMMs do not involve weights, and are typically computed as dense. For dense computation, only one B bank is sufficient at each vertical chain, *e.g.*, $B_{x,0}$ (Fig. 4b). However, multiplexing logic can be employed to utilize the remaining $B_{x,1}$, $B_{x,2}$, $B_{x,3}$ banks. This is particularly important to ensure efficient utilization of on-chip memory resources, which leads to maximized data reuse and thus optimized energy efficiency [71, 72]. Finally, for 1:3 sparsity, only banks $B_{x,0}$, $B_{x,1}$, $B_{x,2}$ are required, leaving bank $B_{x,3}$ unused. However, similar to the dense operation, multiplexing logic can be employed to reuse this bank when it comprises multiple BRAMs, which we do not explore in this work (see Sec. 4.4 for implementation details).

The dynamic configuration among all sparsity levels is implemented in the control logic (using CLBs). However, FPGA accelerators can be designed in a custom fashion depending on the sparsity of each DNN. For instance, a specific DNN might require only 1:3 sparsity and dense computation across all of its layers. The control and tiling logic for sparsity is similar to the dense design (Fig. 4a), showcasing a marginal increase in CLB resources (see Sec. 4.4).

4 Evaluation

4.1 FPGA Architecture

The traditional FPGA architecture comprises CLBs, BRAMs, DSP slices and routing resources, *i.e.*, connection and switch boxes (CBs and SBs). We enrich the FPGA architecture with our proposed in-fabric SST slices. The SST columns repeat every 15 FPGAs columns, occupying only 14% of the total FPGA area, which is sufficient for DNN applications as also found in [13] (refer to Fig. 8 for the layout of the proposed FPGA architecture). We exploit the automated transistor sizing tool, COFFE [17, 84], to model the delays and areas of the FPGA components. These delays and areas are subsequently used to describe the FPGA architecture in the Verilog-to-Routing (VTR) tool flow [61], facilitating FPGA architecture exploration. We utilize the 7nm FinFET ASAP7 predictive process design kit (PDK) [23], for the SPICE simulations conducted by COFFE. Specifically,

Table 3: Area and freq. of SST vs. SDT_GIO slices (post PnR).

	In-fabric slice	SST	SDT_GIO
Area (μm^2)	Standard-cell core	4530.8 (+22.9%)	3687.8
	Input crossbar	1511.4	985.6
	Dedicated output routing	40.2	0
	Switch Box (SB)	1374.3	1603.4
	Connection Box (CB)	734.9	571.6
	Total tile	8191.6 (+19.6%)	6848.4
Freq. (MHz)	int8	928.6	935.3
	bfloat16	838.1	850.5

we exploit the typical corner (TT) ASAP7 model [23, 76]. Finally, COFFE was run with cost function of $area \times delay$ for four transistor sizing iterations (2–4 iterations are typically sufficient [22]).

We model a modern Agilex-like FPGA similar to [13, 14]. Table 2 shows the FPGA logic (CLB) and routing (CBs and SBs) architectural parameters used in COFFE. Each CLB contains 10 fracturable logic elements (FLEs), which consist of 6-input lookup tables (LUTs), registers and two adder bits. Since COFFE does not model 7nm optimized memory circuits, we obtain the model of the 20Kbit BRAMs used in the recent work [17] for 7nm FPGAs. The delay and area values of these BRAMs are conservatively estimated using the 14nm Stratix 10 architecture values, as described in [17]. Finally, we leverage the DSP used in [13], which supports multiple precisions and modes, closely matching the commercial Agilex DSP [38]. However, since this DSP is modeled in 22nm, we scale down its area and delay values to 7nm, exploiting the scaling factors from [69].

4.2 SST Slices Implementation

We implement the SST slices utilizing the ASAP7 7.5-track standard cell library [1, 76]. Specifically, we use the regular threshold voltage (RVT) cells of the typical (TT) corner of the library. COFFE’s hybrid flow was exploited, where the core of the SSTs is implemented using the standard cell flow, while the interface to the programmable routing (local crossbar, drivers for dedicated wires, etc.) is implemented in the full custom flow using SPICE simulations. The standard cell flow uses Synopsys Design Compiler for synthesis, Cadence Innovus for PnR and Synopsys Primetime for timing analysis. The full custom flow uses the TT ASAP7 SPICE model and COFFE was run with the same configurations as in the previous section.

To accurately quantify the area overhead of the sparsity enhancements described in Sec. 3.3, we remove the sparsity features of the SST slices, to implement a 2D systolic dense tensor slice. We retain all the design features delineated in Sec. 3.2, except the dedicated wires. For this dense tensor slice, we utilize the approach proposed in [13, 14], where all I/O ports are connected to global routing. We refer to this tensor slice as SDT_GIO, which is similar to [13, 14]. The dense SDT_GIO slices are used as the baseline comparison with our proposed sparse SSTs, while also allowing for quantification of the routing savings due to dedicated wires in SSTs (Sec. 4.3).

Table 3 shows the area and frequency results for both the SST and SDT_GIO slices obtained from COFFE. At the standard-cell core level, we observe a low area overhead of 22.9% for the sparsity enhancements of the SST slices. For both SST and SDT_GIO slices, we implement a 50% populated input crossbar to enhance their routability inside the FPGA fabric [84]. Each SST slice has 333 global routing inputs and 202 outputs, along with 256 dedicated inputs/outputs. In contrast, the SDT_GIO slice has 259 global routing inputs and 258 outputs, without dedicated I/Os. Given the

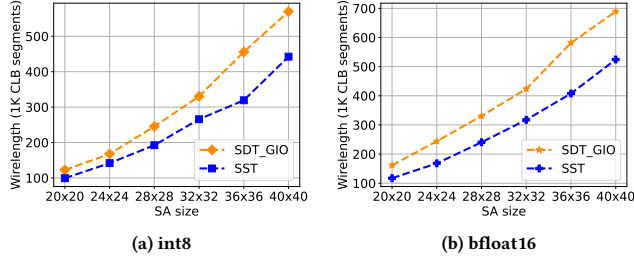


Figure 5: Routing wirelength comparison of dense GEMM mapped to SST and SDT_GIO slices, for various SA sizes.

forementioned global routing I/Os, the SST slices need to access 6 SBs, thus spanning 6 CLB (logic) tiles. However, since more outputs are needed for the SDT_GIOs (primarily due to absence of dedicated wires), they need to span 7 CLB tiles. Therefore, when calculating the total tile area for both slices (standard-cell core and routing interface), we obtain a low area increase of **19.6%** for SSTs vs. SDT_GIOs (Table 3). Finally, we observe that both slices can operate at high frequencies for the supported precisions (> 838 MHz in all cases), similar to commercial 7nm FPGA hard blocks [12, 40].

4.3 Wirelength Gains Utilizing Dedicated Wires

To quantify the benefits of the proposed dedicated wires in SSTs, we implement dense GEMMs utilizing our SSTs slices (Fig. 4a) and compare them with dense GEMMs using SDT_GIOs slices, which support only global routing. To this end, similar to the enhanced FPGA architecture with SST slices, we create an architecture that has SDT_GIOs instead of SSTs. To facilitate the exploration, we implement a parametric Python script that generates the Verilog code for arbitrary GEMM sizes. The Verilog designs are synthesized and implemented on the two aforementioned FPGA architectures (one with SST and one with SDT_GIO slices), exploiting the VTR flow. In all cases, we perform a 10 seed-sweep in VTR, and report the design that attains the maximum FPGA frequency.

Fig. 5 illustrates the total routing wirelength of the two dense GEMM accelerators, for various SA sizes at both int8 and bfloat16 precisions. These designs use 512-element deep banks for buffers A, B and C (Sec. 3.4), exploiting the 512x40-bit BRAM configuration mode (also found in recent Intel FPGAs [39, 42, 43]). For the SA sizes shown in Fig. 5, we observe a significant wirelength reduction due to dedicated wires in the SSTs, ranging from **15.5–29.9%** for int8 precision, compared to the SDT_GIO-based GEMM designs. Similarly, for bfloat16, the wirelength reduction ranges from **23.9–31.2%**, showcasing the importance of vertical dedicated wires for 2D GEMM implementations on FPGA architectures.

Regarding the maximum attainable FPGA frequency, both dense GEMM implementations achieve high frequencies ranging from 668–731 MHz. In particular, the smallest designs, *i.e.*, 20x20 SA size, attain the highest frequency of ~ 731 MHz for both SSTs and SDT_GIOs at int8 and bfloat16 precisions. Similarly, the largest designs, *i.e.*, 40x40 SA size, both attain ~ 668 MHz. We notice that dense GEMMs mapped to SST and SDT_GIO slices attain almost similar FPGA frequencies ($< 1\%$ difference for all tested SA sizes depicted in Fig. 5). This can be explained by the fact that the critical path is typically in the control logic implemented in CLBs, which is the same for both dense GEMMs mapped to SSTs and SDT_GIOs.

4.4 Sparse SST-based GEMM Implementation

In this section, we present the *sparse* GEMM implementation shown in Fig. 4b, which allows dynamic configuration for all supported sparsity modes in SSTs, *i.e.*, dense, 2:4, 1:3 and 1:4. We use two baselines for our comparison. First, we compare with a GEMM design mapped to SDT_GIOs using a FPGA architecture that replaces SSTs with SDT_GIOs. Since SDT_GIOs have a 2D dense systolic dataflow that does not support sparse computation (Sec. 4.2), we retain the zeros of the sparse matrices, therefore operating similarly to dense. Hence, this implementation is similar to that shown in the previous section (Sec. 4.3), however it has the same amount of on-chip memory as the sparse SST-based design. In particular, we implement four memory banks in the vertical dimension to feed each SDT_GIO (similar to Fig. 4b). This allows for the same data reuse opportunities as in the SST-based design, thus enabling a fair comparison. Second, we implement the same sparse SST-based design, mapped to CLBs and DSPs of a traditional FPGA architecture, *i.e.*, without SSTs and SDT_GIOs. Finally, we note that all tested designs utilize 512-element deep banks and are based on a 10 seed-sweep maximization of FPGA frequency, similar to Sec. 4.3.

4.4.1 FPGA Frequency. Fig. 6 depicts the FPGA frequency of the sparse GEMMs mapped to SST slices, SDT_GIO slices as well as CLBs and DSPs. First, we observe the high FPGA frequency of GEMMs mapped to SST and SDT_GIO slices, ranging from 596–625 MHz for int8 and from 578–610 MHz for bfloat16. We note that the slightly lower frequencies of the SDT_GIO-based GEMMs compared to the previous section are attributed to the higher BRAM usage. For all tested SA sizes, we notice a negligible difference in FPGA frequency ($< 1\%$) between the SST-based and the SDT_GIO-based GEMMs. However, when comparing the SST-based vs. the CLB+DSP GEMMs, we observe considerably higher frequencies, ranging from **2.8–4.4 \times** and from **2.7–5 \times** for int8 and bfloat16, respectively.

Second, note that the sparse GEMMs utilizing SSTs scale effectively when increasing the SA size. In particular, we observe a minor decrease in FPGA frequency (4.6% and 5.2% for int8 and bfloat16, respectively), when considering the 40x40 vs. the 20x20 SA size. In contrast, the sparse GEMM mapped to CLBs and DSPs, does not scale as effectively, since we notice a substantial frequency drop of 36.6% and 49.6% for int8 and bfloat16, respectively. These results emphasize the significant performance advantages and scalability of in-fabric sparse hard blocks compared to traditional FPGAs.

4.4.2 GEMM Area. We calculate the total area of our GEMM designs as the summation of the utilized logic area and the FPGA routing area. While the utilized logic area is reported in VTR, the tool does not report the used routing area. Thus, we estimate the routing area by summing the area of all utilized multiplexers in both the SB and CB resources of our designs. Fig. 7 shows the total area of our sparse GEMM designs in minimum width transistor area (MWTAs) units [61]. As shown in Table 3, we observed a 19.6% area overhead of the SSTs vs. the SDT_GIOs at the *tile level*. However, here we calculate this area overhead at the *GEMM implementation level*, as this provides more accurate estimations for DNN accelerators. When comparing the sparse GEMM mapped to SSTs vs. the GEMM using SDT_GIOs, we observe a small area increase ranging from **10.2–15.9%** for int8, and from **13.3–19.4%** for bfloat16, across

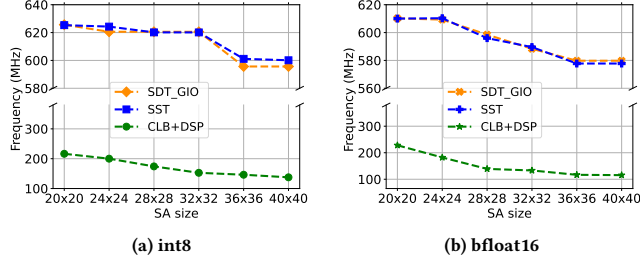


Figure 6: FPGA frequency comparison of *sparse* GEMM using SSTs vs. SDT_GIOs vs. CLBs+DSPs, for various SA sizes.

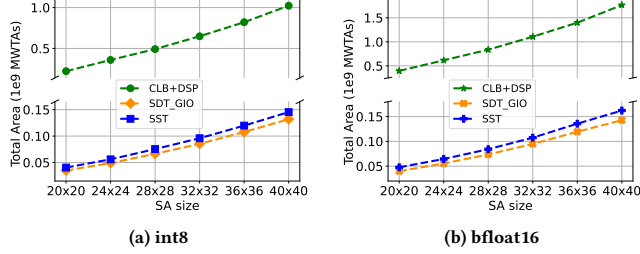


Figure 7: Total area comparison of various *sparse* GEMM implementations utilizing SSTs vs. SDT_GIOs vs. CLBs+DSPs, all SA sizes.

We note that the lowest area increase, *i.e.*, 10.2% and 13.3% for int8 and bfloat16, respectively, occurs for the highest, 40x40 SA size. This is mainly attributed to the dedicated wires in SSTs, where the increase of the SA size (thus the total slices), leads to more routing area in the GEMM design using SDT_GIOs.

As shown in Fig. 7, we notice a substantial area increase of the sparse GEMM mapped to CLBs and DSPs vs. using SSTs, of up to 7 \times for int8, and 10.9 \times for bfloat16. Moreover, we observe that this area difference increases with the SA size. In particular, for 20x20 SA size, we obtain a difference of 5.5 \times and 8.3 \times for int8 and bfloat16, respectively. However, for 40x40 size, the difference increases to 7 \times and 10.9 \times for int8 and bfloat16, respectively. Hence, our results exhibit the significant area efficiency and scalability of the proposed sparse GEMM designs using in-fabric SSTs, over traditional FPGAs.

4.4.3 Resource Usage & Efficiency of Sparse SST-based GEMM. Table 4 presents the resource usage and several evaluation metrics of the sparse GEMM designs, for SA size of 40x40. First, we notice a small CLB increase for the SST vs. the SDT_GIO GEMM, of 26.2% and 29.1% for int8 and bfloat16, respectively. This is ascribed to the additional control logic of including all supported sparsity levels, compared to only control logic for dense (since for SDT_GIOs the zeros of the sparse matrices are retained, effectively operating similar to dense). Second, note that the BRAM usage is the same in all GEMM designs, despite the additional bandwidth requirement due to the indices of the compressed sparse format (Sec. 3.4). This is due to the use of the 512x40-bit BRAM mode (Sec. 4.3). For example, for int8, buffer A banks in the dense design of Fig. 4a, need a bitwidth of 4-8=32-bits, which is smaller than the 40-bits width of the BRAMs. However, for the sparse operation of Fig. 4b, they need additionally 8-bits for the indices (40-bits total), which is exactly the same as the BRAMs bitwidth. Similar conclusions can be drawn for bfloat16. Hence, no additional BRAMs are needed for the higher bandwidth requirements of sparse matrices in compressed format.

Table 4 shows the *effective* throughput and area efficiency of our GEMM designs. The *effective* throughput (TOPs) is calculated on

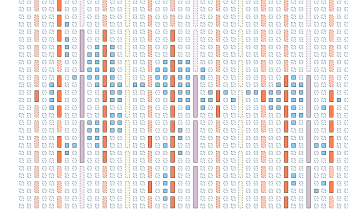


Figure 8: SST-based GEMM implemented in VTR (SA size: 12x12). Blue: CLB, Orange: BRAM, Yellow: DSP, Purple: SST.

the GEMM *native* size (40x40 in this case, see Sec. 3.4), and similar to [7, 60, 63] when acceleration can be attained from sparsity. In particular, since 1:4 sparsity is included in our SST-based and CLB+DSP designs, their *effective* throughput is increased by a factor of four. When comparing the SST-based GEMM vs. the SDT_GIO-based design, we observe a throughput gain of 4.03 \times and 3.98 \times for int8 and bfloat16, respectively. Moreover, we observe 3.66 \times (int8) and 3.51 \times (bfloat16) higher area efficiency (TOPs/GMWTA, area in Giga MWTA). In addition, when comparing with the GEMM using CLBs and DSPs, we obtain 4.34 \times (int8) and 5 \times (bfloat16) higher throughput for the SST-based GEMM. Regarding the area efficiency, an immense 30.62 \times (int8) and 54.45 \times (bfloat16) difference is observed, primarily due to the increased usage of the traditional FPGA resources. To this end, notice the very high CLB and wirelength usage for the CLB+DSP vs. the SST-based GEMM. For instance, for int8, there is a 34.1 \times higher CLB and a 15 \times higher wirelength usage.

Finally, in Fig. 8 we show a screenshot of a sparse SST-based design implemented in VTR. Notice the physically contiguous vertical SST chains due to dedicated wires. Also, observe the 2D *physical* layout after VTR PnR, compared to the *logical* 2D array of Fig. 4b.

4.5 DNN Speedup Estimation Using SST Slices

We provide a speedup estimation for actual DNNs when leveraging our proposed SST slices. In particular, we utilize the DeiT [74] and ConvNeXt [54] models, which attain state-of-the-art accuracy for vision tasks. We apply two types of sparsity to determine the optimal sparsity configuration for each aforementioned models. First, we utilize *uniform* sparsity, *i.e.*, all layers have the same sparsity, by pruning weights based on their magnitudes [60]. Second, we exploit *layer-wise* sparsity by adopting the neural architecture search (NAS) methodology proposed in [36] to identify the optimal sparsity level of each layer, aiming to minimize accuracy degradation. Accuracy is evaluated on the ImageNet-1K dataset [24], with int8 quantization applied using the post-training method proposed in [86].

For speedup estimation, we develop an analytical model based on our results in Sec. 4.4, comparing the sparse SST-based GEMM to the SDT_GIO-based GEMM, with the latter operating on dense matrices. Our model exploits the matrix sizes for each layer of the DeiT and ConvNeXt models, and applies zero padding to align with the *native* size of the GEMM accelerator (Sec. 3.4). We utilize our largest implemented GEMM design, *i.e.*, 40x40 SA size, and assume 100 GB/s bandwidth for DRAM modeling, which is typical in modern FPGAs [3, 10]. Utilizing GEMM computations provides a reliable model for estimating speedup, since GEMM is the core computation in contemporary DNNs, accounting for more than 90% of the total execution time [4, 79]. Other operations *e.g.*, softmax, layer-norm, can be effectively overlapped by DNN accelerators. Moreover, GEMM-based estimation (convolutions implemented as GEMM for

Table 4: Evaluation and comparison of sparse GEMM designs using SSTs vs. SDT_GIOs vs. CLBs+DSPs (SA size: 40x40).

Pr.	GEMM Cfg.	CLBs	BRAMs	DSP slices	SST slices	SDT_GIO slices	Freq. (MHz)	Area (GMWTA)	Wirelength (CLB seg. units)	Eff. Thrpt. (TOPs)	Area Eff. (TOPs/GMWTA)
int8	SST	665	450	0	100	–	601	0.145	493K	7.69	53.03
	SDT_GIO	527	450	0	–	100	596	0.132	578K	1.91	14.47
	CLB+DSP	22650	450	400	–	–	138	1.022	7373K	1.77	1.732
bfloat16	SST	719	500	0	100	–	578	0.162	684K	7.40	45.68
	SDT_GIO	557	500	0	–	100	580	0.143	725K	1.86	13.01
	CLB+DSP	27065	500	2400	–	–	116	1.764	15212K	1.48	0.839

Table 5: Speedup estimation, accuracy and weight mem. reduction of various DNNs when mapped to SSTs vs. SDT_GIOs.

DNN model	Sparsity levels	Number of layers			In-fabric slice	Speedup	Top-1 accuracy (%)		Weight reduction		
		Dense	2:4	1:3			1:4	int8	bfloat16	int8	bfloat16
DeiT-S (4.7 GFLOPs)	dense	48	–	–	–	SDT_GIO	1×	79.56 (-0.00%)	79.79 (-0.00%)	1×	1×
	[dense, 2:4]	–	48	–	–	SST	1.88×	79.08 (-0.48%)	79.20 (-0.59%)	1.58×	1.76×
	[dense, 2:4, 1:3, 1:4]	3	26	16	3	SST	2.14×	77.95 (-1.61%)	78.33 (-1.46%)	1.81×	2.02×
	[dense, 1:3]	–	–	48	–	SST	2.61×	73.36 (-6.20%)	73.95 (-5.84%)	2.32×	2.61×
DeiT-B (17.6 GFLOPs)	dense	48	–	–	–	SDT_GIO	1×	81.40 (-0.00%)	81.80 (-0.00%)	1×	1×
	[dense, 2:4]	–	48	–	–	SST	1.91×	81.45 (+0.05%)	81.73 (-0.07%)	1.59×	1.77×
	[dense, 2:4, 1:3, 1:4]	6	15	13	14	SST	2.35×	81.28 (-0.12%)	81.60 (-0.20%)	2.05×	2.26×
	[dense, 1:3]	–	–	48	–	SST	2.75×	81.10 (-0.30%)	81.25 (-0.55%)	2.36×	2.64×
	[dense, 1:4]	–	–	–	48	SST	3.52×	80.37 (-1.03%)	80.69 (-1.11%)	3.12×	3.50×
ConvNeXt-S (8.7 GFLOPs)	dense	108	–	–	–	SDT_GIO	1×	82.99 (-0.00%)	83.09 (-0.00%)	1×	1×
	[dense, 2:4]	36	72	–	–	SST	1.93×	82.52 (-0.47%)	82.62 (-0.47%)	1.46×	1.66×
	[dense, 2:4, 1:3, 1:4]	36	59	7	6	SST	2.07×	82.30 (-0.69%)	82.39 (-0.70%)	1.52×	1.74×
	[dense, 1:3]	36	–	72	–	SST	2.81×	81.44 (-1.55%)	81.53 (-1.56%)	1.95×	2.32×
	[dense, 1:4]	36	–	–	72	SST	3.63×	81.05 (-1.94%)	81.16 (-1.93%)	2.34×	2.89×

ConvNeXt) is also performed in multiple works targeting speedup calculations due to sparsity exploitation [47, 55, 75, 81, 91].

Table 5 presents the speedup estimation and accuracy for our two ViT models, *i.e.*, DeiT-S and DeiT-B, and the ConvNeXt-S model. When applying uniform 2:4 sparsity, we observe a negligible accuracy decrease in DeiT-B for bfloat16 (0.07%), or even slightly higher accuracy for int8 (by +0.05%). However, we notice a larger decrease for DeiT-S, *e.g.*, 0.48% for int8. This is due to the fact that smaller models (DeiT-S with 4.7 GFLOPs) exhibit less redundancy compared to larger models (DeiT-B with 17.6 GFLOPs), making them more resilient to sparsity. Note that even with uniform 2:4 sparsity across all layers, dense computation is still needed (Table 5), due to QKV computation in ViTs (Sec. 3.4). In addition, for ConvNeXt-S, we retain its depth-wise convolution layers as dense, since we observed a severe accuracy degradation when attempting to sparsify them. For instance, when applying 2:4 sparsity to all other layers except the (36) depth-wise layers in ConvNeXt-S, we obtain an accuracy degradation of only 0.47%. However, uniform 2:4 sparsity can result in significant speedup, *e.g.*, 1.93× in ConvNeXt-S, compared to dense operation. We note that both int8 and bfloat16 cases deliver nearly indistinguishable speedups over dense computation, since the frequency difference between the SST-based and SDT_GIO-based GEMM accelerators is less than 1% (Sec. 4.4).

When applying layer-wise sparsity (all supported levels), we observe higher speedup in all cases, *e.g.*, 2.35× in DeiT-B. In this case, for DeiT-B, accuracy degradation is still negligible, *e.g.*, 0.12%. However, we observe higher degradation for DeiT-S (1.61% for int8) and ConvNeXt-S (0.69% for int8), since they are both small models. When applying higher sparsity (uniform 1:3 and 1:4), accuracy degrades even further, but higher speedup is attained, showcasing the accuracy-speedup trade-off. If we constraint the accuracy degradation to be ~1% (acceptable in the vast majority of applications [70]), the optimal speedup is attained for different sparsity configurations

in DNNs (bolded in Table 5). Specifically, for DeiT-S, uniform 2:4 is optimal (**1.88×** speedup), while for DeiT-B, uniform 1:4 shows best results (**3.52×** speedup). For ConvNeXt-S, layer-wise sparsity (all supported levels) exhibits optimal results (**2.07×** speedup). However, when allowing higher accuracy degradation (for instance, within 2%) higher speedup can be attained, *e.g.*, 3.63× for uniform 1:4 in ConvNeXt-S. Finally, notice the higher weight memory reduction as sparsity increases (Table 5). Our results demonstrate the importance of supporting multiple sparsity levels in the SST slices.

4.6 Versal AIE-ML Sparsity Comparison

The new Versal AIE-ML [8, 11] includes out-of-fabric processors that support 2:4 sparsity. Therefore, we aim to compare the attainable acceleration due to sparsity and the efficiency of the compressed sparse format between our SSTs and the AIE-ML. We exploit AMD’s *optimized* codes for dense/sparse GEMM kernels for the AIE-ML [5, 6], using 16-bits for the outputs as this leads to substantially higher compute utilization [6]. We compile and simulate the AIE-ML designs using Vitis 2023.2 on the VEK280 platform [9].

Fig. 9a shows the compute utilization of a 64×64×64 GEMM, mapped to both SST slices and the AIE-ML. We notice that the AIE-ML achieves higher than 90% utilization in dense GEMM for both precisions. However, for 2:4 sparsity, we observe a substantially lower utilization, *i.e.*, 51.6% for int8 and 50.4% for bfloat16. We note that the compute utilization is calculated as the *effective* utilization of only non-zero computation in the sparse case. Thus, we obtain a marginal speedup of 13% (int8) and 4.5% (bfloat16), for the 2:4 sparse GEMM over the dense case for the AIE-ML. Moreover, for higher sparsity, *i.e.*, 1:3 and 1:4, the compute utilization in AIE-ML drops more significantly (Fig. 9a). While higher than 2:4 sparsity can be stored in a compressed format in the AIE-ML, only 2:4 sparsity is inherently supported. Consequently, no further acceleration can be achieved, leading to significantly decreased *effective* utilization.

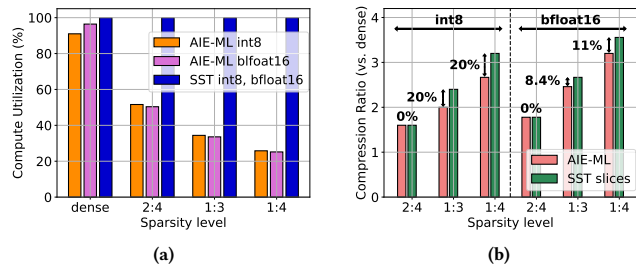


Figure 9: Compute utilization (a) and compression ratio (b) comparison of a $64 \times 64 \times 64$ GEMM using SSTs vs. AIE-ML.

This marginal speedup due to sparsity in the AIE-ML is primarily attributed to limited vector load/store bandwidth, causing the sparse GEMM to become I/O bound [6, 11]. In contrast, as depicted in Fig. 9a, our SST-based GEMM achieves 100% utilization for all supported sparsity levels. This is because we architect our SST slices in a fashion that guarantees the corresponding speedup of each sparsity level, e.g., 4 \times for 1:4 sparsity (refer to Sec. 3.3 for design details).

Fig. 9b shows the compression ratios achieved by the SSTs vs. the AIE-ML. The AIE-ML uses a bitmap-based compressed format [8], which is also used in various works accelerating structured sparsity [18, 27, 55, 57, 83]. Instead, we utilize an index-based format (Sec. 3.1). For 2:4 sparsity, both approaches present the same compression ratio (Fig. 9b). However, for 1:3 and 1:4, our approach leads to higher compression ratio, up to 20% for int8 and 11% for bfloat16, showcasing the storage efficiency of the index-based format. Moreover, as shown in [27], bitmap achieves higher compression ratio over traditional sparse formats, e.g., CSR, CSC, COO [35], for sparsity levels of 50–87.5%. This highlights that the supported index-based sparse format provides superior compression over other formats.

4.7 Impact on Non-AI Benchmarks

We demonstrate the flexibility of our enhanced FPGA architecture with SST slices, by exploring the maximum attainable frequency of non-AI benchmarks. We select non-AI benchmarks from the VTR benchmark suite [61], which target various application domains, i.e., *sha*, *mmcl*, *arm_core*, *stereovision2*, *LU32PEEng*, *bgm*, *raygentop*, and *blob_merge* (see [78] for domains). When comparing our enhanced FPGA with a traditional FPGA on the aforementioned benchmarks, we observe, on average, a negligible decrease (<1%) in maximum frequency. Hence, our proposed FPGAs maintain the performance and flexibility of traditional FPGAs, while offering highly efficient sparse and dense DNN acceleration (Sec. 4.3–4.5).

4.8 Insights & Discussion

4.8.1 Very High Sparsity in DNN Layers. In the previous sections, we exhibit the importance and efficiency of our proposed in-fabric SST slices for accelerating sparse DNNs. However, our SSTs support up to 75% (1:4) structured sparsity. Prior work [32, 33] has shown that a few DNN layers can have very high *random* sparsity, e.g., >90%. In this case, *unstructured* sparse FPGA accelerators mapped to traditional CLB and DSP resources, e.g., [28, 46, 59], can be optimized to efficiently compute these few high-sparsity layers. All other DNN layers with less sparsity can be mapped to our SSTs (Sec. 4.4 & 4.5), enabling both *structured* and *unstructured* sparse FPGA accelerators to operate synergistically, for maximized efficiency.

Moreover, for very high sparsity (>90%), the CSR, CSC, COO formats utilized in unstructured sparse accelerators, offer the most efficient compression [35]. Instead, for lower sparsity, the index-based format utilized in SSTs is the most efficient (Sec. 4.6). This emphasizes the custom flexibility of the enhanced FPGAs with in-fabric SST slices in supporting highly efficient sparse DNN accelerators.

4.8.2 Efficient In-Fabric Slices. In-fabric slices have attained commercial success in AI-optimized FPGAs, e.g., the Intel tensor blocks [51] and the new Intel AI-enhanced DSPs [30, 41]. Moreover, recent research [72] comparing leading in-fabric (Intel Stratix 10 NX) with out-of-fabric (AMD Versal ACAP) AI-optimized FPGAs, has shown higher energy efficiency in GEMM for in-fabric solutions. However, these efficient in-fabric slices primarily target dense AI acceleration, rendering them insufficient for most DNNs, which are sparse. Instead, our proposed flexible SST slices support both dense and sparse computation (multiple sparsity levels), showcasing substantial advantages over dense in-fabric blocks and traditional FPGAs (Sec. 4.4), for actual sparse DNN workloads (Sec. 4.5). Note that although we are inspired by prior academic research that utilize dense 2D systolic in-fabric blocks [13, 14], the methodology described in Sec. 3 for introducing sparsity support can be generalized in straightforward fashion to other in-fabric FPGA blocks.

4.8.3 Future Directions. Our SST slices support *static* sparsity targeting weights in DNNs. Future work could also explore supporting *dynamic* sparsity in activations. The SST slices could also enable other modes for increased flexibility, e.g., element-wise modes, similar to [13, 14]. However, this is beyond the scope of this paper and is therefore left for future work. We note that general matrix-vector (GEMV) can be directly supported in SSTs at batch size of four with 100% utilization (by setting $X=1$ in designs of Sec. 3.4). Additionally, multiplexing logic can be incorporated into SSTs to increase GEMV utilization for batch size of one, as shown in [13], which we leave as future work. Finally, another extension is to include multiple input crossbars, which would further reduce the small area overhead of the SSTs, and quantify the trade-offs in area and FPGA routability.

5 Conclusion

Structured sparsity support has been incorporated in state-of-the-art GPUs, e.g., Nvidia H100 and AMD MI300, as well as the new Versal AIE-ML processors. Although FPGA architectures have been enriched with in-fabric blocks for DNN acceleration, these blocks are primarily designed for dense operation. This leads to insufficient computation for most contemporary DNN models, which display varying levels of sparsity. To effectively address this challenge, we propose flexible in-fabric blocks, named SST slices, that support multiple levels of structured sparsity. We show that our sparse GEMM accelerators exploiting the SST slices offer substantial performance, scalability, and area advantages over traditional FPGAs. Demonstration on various state-of-the-art sparse DNN models utilizing our SST slices, exhibits up to 3.52 \times speedup with marginal accuracy degradation compared to dense in-fabric acceleration.

Acknowledgment

This work was supported by the National Science Foundation CCF Grant No. 2107085, the ONR Minerva program, and iMAGINE – the Intelligent Machine Engineering Consortium at UT Austin.

References

- [1] 2023. ASAP7 PDK and Cell Libraries. <https://github.com/The-OpenROAD-Project/asap7>.
- [2] Achronix. 2024. Machine Learning Processor. <https://www.achronix.com/machine-learning-processor>.
- [3] Achronix. 2024. Speedster7t FPGAs: Product Brief. https://www.achronix.com/sites/default/files/docs/Speedster7t_Product_Brief_PB033.pdf.
- [4] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2016. Fathom: Reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–10.
- [5] AMD. 2023. AI Engine API User Guide. https://www.xilinx.com/htmldocs/xilinx2023_2/aiengine_api/aiengine_api/doc/group_group_mmul.html.
- [6] AMD. 2023. AI Engine-ML Programming. https://github.com/Xilinx/Vitis-Tutorials/blob/2023.2/AI_Engine_Development/AIE-ML/Design_Tutorials/01-AIE-ML-programming-and-optimization/ComputeOptimization.md.
- [7] AMD. 2023. AMD CDNA 3 Architecture. <https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf>.
- [8] AMD. 2024. AI Engine-ML Kernel and Graph Programming Guide (UG1603). <https://docs.amd.com/r/en-US/ug1603-ai-engine-ml-kernel-graph/Overview?tocId=UXKuGgdu4z0LghCWyYalPw>.
- [9] AMD. 2024. AMD Versal™ AI Edge Series VEK280 Evaluation Kit. <https://www.xilinx.com/products/boards-and-kits/vek280.html>.
- [10] AMD. 2024. VCK5000 Versal Development Card. <https://www.xilinx.com/products/boards-and-kits/vck5000.html>.
- [11] AMD. 2024. Versal Adaptive SoC AIE-ML Architecture Manual (AM020). <https://docs.amd.com/r/en-US/am020-versal-ai-ml/Overview>.
- [12] AMD. 2024. Versal AI Core Series Data Sheet: DC and AC Switching Characteristics (DS957). <https://docs.amd.com/r/en-US/ds957-versal-ai-core/DSP58-Switching-Characteristics>.
- [13] Aman Arora, Moinak Ghosh, Samidh Mehta, Vaughn Betz, and Lizy K. John. 2022. Tensor Slices: FPGA Building Blocks For The Deep Learning Era. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 46 (aug 2022), 34 pages. <https://doi.org/10.1145/3529650>
- [14] Aman Arora, Samidh Mehta, Vaughn Betz, and Lizy K. John. 2021. Tensor Slices to the Rescue: Supercharging ML Acceleration on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Virtual Event, USA) (FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 23–33. <https://doi.org/10.1145/3431920.3439282>
- [15] Andrew Boutros, Aman Arora, and Vaughn Betz. 2024. Field-Programmable Gate Array Architecture for Deep Learning: Survey & Future Directions. arXiv:2404.10076 [cs.AR] <https://arxiv.org/abs/2404.10076>
- [16] Andrew Boutros and Vaughn Betz. 2021. FPGA Architecture: Principles and Progression. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 4–29. <https://doi.org/10.1109/MCAS.2021.3071607>
- [17] Andrew Boutros, Fatemehsadat Mahmoudi, Amin Mohaghegh, Stephen More, and Vaughn Betz. 2023. Into the Third Dimension: Architecture Exploration Tools for 3D Reconfigurable Acceleration Devices. In *2023 International Conference on Field Programmable Technology (ICFPT)*. 198–208. <https://doi.org/10.1109/ICFPT59805.2023.00027>
- [18] Lei Cai, Jing Wang, Lianfeng Yu, Bonan Yan, Yaoyu Tao, and Yuchao Yang. 2023. Accelerating Neural-ODE Inference on FPGAs with Two-Stage Structured Pruning and History-based Stepsize Search. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (Monterey, CA, USA) (FPGA '23)*. Association for Computing Machinery, New York, NY, USA, 177–183. <https://doi.org/10.1145/3543622.3573044>
- [19] Aiken Cairncross, Basile Henry, Chris Chalmers, Douglas Reid, Jonny Shipton, Jon Fowler, Liz Corrigan, and Mike Ashby. 2023. AI Benchmarking on Achronix Speedster™ 7t FPGAs. (2023).
- [20] Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. 2019. Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Seaside, CA, USA) (FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 63–72. <https://doi.org/10.1145/3289602.3293898>
- [21] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. <https://doi.org/10.1109/JETCAS.2019.2910232>
- [22] Charles Chiasson and Vaughn Betz. 2013. COFFE: Fully-automated transistor sizing for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT)*. 34–41. <https://doi.org/10.1109/FPT.2013.6718327>
- [23] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016), 105–115. <https://doi.org/10.1016/j.mejo.2016.04.006>
- [24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. Ieee, 248–255.
- [25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *North American Chapter of the Association for Computational Linguistics*. <https://api.semanticscholar.org/CorpusID:52967399>
- [26] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xi-aohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2020. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *ArXiv abs/2010.11929* (2020). <https://api.semanticscholar.org/CorpusID:225039882>
- [27] Chao Fang, Aojun Zhou, and Zhongfeng Wang. 2022. An Algorithm–Hardware Co-Optimized Framework for Accelerating N:M Sparse Transformers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30, 11 (2022), 1573–1586. <https://doi.org/10.1109/TVLSI.2022.3197282>
- [28] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. 2014. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 36–43. <https://doi.org/10.1109/FCCM.2014.23>
- [29] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (Columbus, OH, USA) (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 151–165. <https://doi.org/10.1145/3352460.3358291>
- [30] Sergey Gribok and Bogdan Pasca. 2024. Efficient 8-bit Matrix Multiplication on Intel Agilex-5 FPGAs. In *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 43–53. <https://doi.org/10.1109/FCCM60383.2024.00016>
- [31] Kai Han, Yunhe Wang, Hanqing Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunqing Xu, Yixing Xu, Zhaohui Yang, Yiman Zhang, and Dacheng Tao. 2023. A Survey on Vision Transformer. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 1 (2023), 87–110. <https://doi.org/10.1109/TPAMI.2022.3152247>
- [32] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine On Compressed Deep Neural Network (ISCA '16). IEEE Press, 243–254. <https://doi.org/10.1109/ISCA.2016.30>
- [33] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (Montreal, Canada) (NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.
- [34] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (Columbus, OH, USA) (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. <https://doi.org/10.1145/3352460.3358275>
- [35] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks. *J. Mach. Learn. Res.* 22, 1, Article 241 (jan 2021), 124 pages.
- [36] Ning-Chi Huang, Chi-Chih Chang, Wei-Cheng Lin, Endri Taka, Diana Marculescu, and Kai-Chiang Wu. 2024. ELSA: Exploiting Layer-wise N:M Sparsity for Vision Transformer Acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. 8006–8015.
- [37] Sitao Huang, Carl Pearson, Rakesh Nagi, Jinjun Xiong, Deming Chen, and Wenmei Hwu. 2019. Accelerating Sparse Deep Neural Networks on FPGAs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. <https://doi.org/10.1109/HPEC.2019.8916419>
- [38] Intel. 2020. Intel Agilex Variable Precision DSP Blocks User Guide. https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/agilex/ug-ag-dsp.pdf.
- [39] Intel. 2023. Intel Stratix 10 Embedded Memory User Guide. <https://www.intel.com/content/www/us/en/docs/programmable/683423/23-2/embedded-memory-configurations.html>.
- [40] Intel. 2024. Agilex™ 5 FPGAs and SoCs Device Data Sheet. <https://www.intel.com/content/www/us/en/docs/programmable/813918/current/agilex-5-fpgas-and-socs-device-data-sheet.html>.
- [41] Intel. 2024. Agilex™ 5 FPGAs: Enhanced DSP with AI Tensor Block. <https://www.intel.com/content/www/us/en/content-details/776602/agilex-5-fpgas-enhanced-dsp-with-ai-tensor-block.html>.
- [42] Intel. 2024. Embedded Memory User Guide: Agilex™ 5 FPGAs and SoCs. <https://www.intel.com/content/www/us/en/docs/programmable/813901/24-1/embedded-memory-configurations.html>.
- [43] Intel. 2024. Embedded Memory User Guide: Agilex™ 7 FPGAs and SoCs. <https://www.intel.com/content/www/us/en/docs/programmable/683241/24-2/embedded-memory-configurations.html>.

- [44] Saidul Islam, Hanae Elmekki, Ahmed Elsebai, Jamal Bentahar, Nagat Drawel, Gaith Rjoub, and Witold Pedrycz. 2024. A Comprehensive Survey on Applications of Transformers for Deep Learning Tasks. *Expert Systems with Applications* 241 (2024), 122666. <https://doi.org/10.1016/j.eswa.2023.122666>
- [45] Abhishek Kumar Jain, Sharan Kumar, Aashish Tripathi, and Dinesh Gaitonde. 2021. Sparse Deep Neural Network Acceleration on HBM-Enabled FPGA Platform. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. <https://doi.org/10.1109/HPEC49654.2021.9622804>
- [46] Abhishek Kumar Jain, Hossein Omidian, Henri Fraise, Mansimran Benipal, Lisa Liu, and Dinesh Gaitonde. 2020. A Domain-Specific Architecture for Accelerating Sparse Matrix Vector Multiplication on FPGAs. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 127–132. <https://doi.org/10.1109/FPL50879.2020.00031>
- [47] Geonhwa Jeong, Sana Damani, Abhimanyu Rajeshkumar Bambhaniya, Eric Qin, Christopher J. Hughes, Sreenivas Subramoney, Hyesoon Kim, and Tushar Krishna. 2023. VEGETA: Vertically-Integrated Extensions for Sparse/Dense GEMM Tile Acceleration on CPUs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 259–272. <https://doi.org/10.1109/HPCA56546.2023.10071058>
- [48] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, Sushma Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Jushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUV4: Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. <https://doi.org/10.1109/ISCA52012.2021.00010>
- [49] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, and et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. <https://doi.org/10.1145/3140659.3080246>
- [50] Kung. 1982. Why systolic architectures? *Computer* 15, 1 (1982), 37–46. <https://doi.org/10.1109/MC.1982.1653825>
- [51] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. 2021. Stratix 10 NX Architecture and Applications. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Virtual Event, USA) (FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 57–67. <https://doi.org/10.1145/3431920.3439293>
- [52] Yun Liang, Liqiang Lu, Yicheng Jin, Jiaming Xie, Ruirui Huang, Jiansong Zhang, and Wei Lin. 2022. An Efficient Hardware Design for Accelerating Sparse CNNs With NAS-Based Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 3 (2022), 597–613. <https://doi.org/10.1109/TCAD.2021.3066563>
- [53] Linqiao Liu and Stephen Brown. 2021. Leveraging Fine-grained Structured Sparsity for CNN Inference on Systolic Array Architectures. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 301–305. <https://doi.org/10.1109/FPL53798.2021.00060>
- [54] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. 2022. A ConvNet for the 2020s. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 11966–11976. <https://doi.org/10.1109/CVPR52688.2022.01167>
- [55] Zhi-Gang Liu, Paul N. Whatmough, and Matthew Mattina. 2020. Sparse Systolic Tensor Array for Efficient CNN Hardware Acceleration. arXiv:2009.02381 [cs.AR] <https://arxiv.org/abs/2009.02381>
- [56] Zhi-Gang Liu, Paul N. Whatmough, and Matthew Mattina. 2020. Systolic Tensor Array: An Efficient Structured-Sparse GEMM Accelerator for Mobile CNN Inference. *IEEE Computer Architecture Letters* 19, 1 (2020), 34–37. <https://doi.org/10.1109/LCA.2020.2979965>
- [57] Zhi-Gang Liu, Paul N. Whatmough, Yuhao Zhu, and Matthew Mattina. 2022. S2TA: Exploiting Structured Sparsity for Energy-Efficient Mobile CNN Acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 573–586. <https://doi.org/10.1109/HPCA53966.2022.00049>
- [58] Liqiang Lu and Yun Liang. 2018. SpWA: An Efficient Sparse Winograd Convolutional Neural Networks Accelerator on FPGAs. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6. <https://doi.org/10.1109/DAC.2018.8465842>
- [59] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. 2019. An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 17–25. <https://doi.org/10.1109/FCCM.2019.00013>
- [60] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stolic, Dusan Stolic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating Sparse Deep Neural Networks. <https://arxiv.org/abs/2104.08378>
- [61] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. 2020. VTR 8: High-performance CAD and Customizable FPGA Architecture Modelling. *ACM Trans. Reconfigurable Technol. Syst.* 13, 2, Article 9 (jun 2020), 55 pages. <https://doi.org/10.1145/3388617>
- [62] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The Design Process for Google's Training Chips: TPUV2 and TPUV3. *IEEE Micro* 41, 2 (2021), 56–63. <https://doi.org/10.1109/MM.2021.3058217>
- [63] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>
- [64] NVIDIA. 2023. Confidential Compute on NVIDIA Hopper H100. <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf>
- [65] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Apoorva Amarnath, Siy-ing Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736. <https://doi.org/10.1109/HPCA.2018.00067>
- [66] SeyedRamin Rasoulzadeh, Hao Zhou, Lingli Wang, and Philip H.W. Leong. 2019. PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 35–44. <https://doi.org/10.1109/FCCM.2019.00015>
- [67] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 58–68. <https://doi.org/10.1109/ISPASS48437.2020.00016>
- [68] Gil Shomron, Tal Horowitz, and Uri Weiser. 2019. SMT-SA: Simultaneous Multithreading in Systolic Arrays. *IEEE Computer Architecture Letters* 18, 2 (2019), 99–102. <https://doi.org/10.1109/LCA.2019.2924007>
- [69] A. Stillmaker and B. Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal* 58 (2017), 74–81. <http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIIntegration.TechScale/>
- [70] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi (Viji) Srinivasan, and Kailash Gopalakrishnan. 2020. Ultra-Low Precision 4-bit Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1796–1807. https://proceedings.neurips.cc/paper_files/paper/2020/file/13b919438259814cd5be8cb45877d577-Paper.pdf
- [71] Endri Taka, Aman Arora, Kai-Chiang Wu, and Diana Marculescu. 2023. MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine. In *2023 International Conference on Field Programmable Technology (ICFPT)*. 96–105. <https://doi.org/10.1109/ICFPT59805.2023.00016>
- [72] Endri Taka, Dimitrios Gourounas, Andreas Gerstlauer, Diana Marculescu, and Aman Arora. 2024. Efficient Approaches for GEMM Acceleration on Leading AI-Optimized FPGAs. In *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 54–65. <https://doi.org/10.1109/FCCM60383.2024.00015>
- [73] V. Titopoulos, K. Alexandridis, C. Peltekis, C. Nicopoulos, and G. Dimitrakopoulos. 2024. IndexMAC: A Custom RISC-V Vector Instruction to Accelerate Structured-Sparse Matrix Multiplications. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. <https://doi.org/10.23919/DAT58400.2024.10546747>
- [74] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Herve Jegou. 2021. Training Data-efficient Image Transformers & Distillation Through Attention. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 10347–10357. <https://proceedings.mlr.press/v139/touvron21a.html>
- [75] Fengbin Tu, Yiqi Wang, Ling Liang, Yufei Ding, Leibo Liu, Shaojun Wei, Shouyi Yin, and Yuan Xie. 2023. SDP: Co-Designing Algorithm, Dataflow, and Architecture for In-SRAM Sparse NN Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 1 (2023), 109–121. <https://doi.org/10.1109/TCAD.2022.3172600>
- [76] Vinay Vashishtha, Manoj Vangala, and Lawrence T. Clark. 2017. ASAP7 predictive design kit development and cell design technology co-optimization: Invited paper. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 992–998. <https://doi.org/10.1109/ICCAD.2017.8203889>
- [77] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (Long Beach, California, USA) (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

- [78] VTR. 2023. Verilog-to-Routing Documentation. <https://docs.verilogtorouting.org/en/latest/vtr/benchmarks/>.
- [79] Yu Emma Wang, Carole-Jean Wu, Xiaodong Wang, Kim Hazelwood, and David Brooks. 2021. Exploiting Parallelism Opportunities with Deep Learning Frameworks. *ACM Trans. Archit. Code Optim.* 18, 1, Article 9 (Dec 2021), 23 pages. <https://doi.org/10.1145/3431388>
- [80] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. <https://doi.org/10.1145/3061639.3062207>
- [81] Yannan Nellie Wu, Po-An Tsai, Saurav Muralidharan, Angshuman Parashar, Vivienne Sze, and Joel Emer. 2023. HighLight: Efficient and Flexible DNN Acceleration with Hierarchical Structured Sparsity. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) (*MICRO '23*). Association for Computing Machinery, New York, NY, USA, 1106–1120. <https://doi.org/10.1145/3613424.3623786>
- [82] Xiaoru Xie, Jun Lin, Zhongfeng Wang, and Jinghe Wei. 2021. An Efficient and Flexible Accelerator Design for Sparse Convolutional Neural Networks. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 7 (2021), 2936–2949. <https://doi.org/10.1109/TCSI.2021.3074300>
- [83] Shuxin Yang, Chenchen Ding, Mingqiang Huang, Kai Li, Chenghao Li, Zikun Wei, Sixiao Huang, Jingyao Dong, Liuyang Zhang, and Hao Yu. 2024. LAMPS: A Layer-wised Mixed-Precision-and-Sparsity Accelerator for NAS-Optimized CNNs on FPGA. In *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 90–96. <https://doi.org/10.1109/FCCM60383.2024.00019>
- [84] Sadegh Yazdanshenas and Vaughn Betz. 2019. COFFE 2: Automatic Modelling and Optimization of Complex and Heterogeneous FPGA Architectures. *ACM Trans. Reconfigurable Technol. Syst.* 12, 1, Article 3 (Jan 2019), 27 pages. <https://doi.org/10.1145/3301298>
- [85] Wenhua Ye, Xu Zhou, Joey Zhou, Cen Chen, and Kenli Li. 2023. Accelerating Attention Mechanism on FPGAs based on Efficient Reconfigurable Systolic Array. *ACM Trans. Embed. Comput. Syst.* 22, 6, Article 93 (Nov 2023), 22 pages. <https://doi.org/10.1145/3549937>
- [86] Zhihang Yuan, Chenhao Xue, Yiqi Chen, Qiang Wu, and Guangyu Sun. 2022. PTQ4ViT: Post-training Quantization for Vision Transformers with Twin Uniform Quantization. In *Computer Vision – ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XII* (Tel Aviv, Israel). Springer-Verlag, Berlin, Heidelberg, 191–207. https://doi.org/10.1007/978-3-031-19775-8_12
- [87] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. <https://doi.org/10.1109/MICRO.2016.7783723>
- [88] Yuxin Zhang, Mingbao Lin, Zhihang Lin, Yiting Luo, Ke Li, Fei Chao, Yongjian Wu, and Rongrong Ji. 2022. Learning Best Combination for Efficient N:M Sparsity. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 941–953. https://proceedings.neurips.cc/paper_files/paper/2022/file/06589ec9d86876508600a678f9c8f51d-Paper-Conference.pdf
- [89] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), 261–274. <https://api.semanticscholar.org/CorpusID:211205022>
- [90] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch. arXiv:2102.04010 [cs.CV] <https://arxiv.org/abs/2102.04010>
- [91] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-wise Sparse Neural Networks on Modern GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) (*MICRO '52*). Association for Computing Machinery, New York, NY, USA, 359–371. <https://doi.org/10.1145/3352460.3358269>