Tensor Slices: FPGA Building Blocks For The Deep Learning Era

AMAN ARORA, University of Texas, USA
MOINAK GHOSH, IIT Kharagpur, India
SAMIDH MEHTA, BITS Pilani, India
VAUGHN BETZ, University of Toronto, Canada
LIZY K. JOHN, University of Texas, USA

FPGAs are well-suited for accelerating deep learning (DL) applications owing to the rapidly changing algorithms, network architectures and computation requirements in this field. However, the generic building blocks available on traditional FPGAs limit the acceleration that can be achieved. Many modifications to FPGA architecture have been proposed and deployed including adding specialized artificial intelligence (AI) processing engines, adding support for smaller precision math like 8-bit fixed point and IEEE half-precision (fp16) in DSP slices, adding shadow multipliers in logic blocks, etc. In this paper, we describe replacing a portion of the FPGA's programmable logic area with Tensor Slices. These slices have a systolic array of processing elements at their heart that support multiple tensor operations, multiple dynamically-selectable precisions and can be dynamically fractured into individual multipliers and MACs (multiply-and-accumulate). These slices have a local crossbar at the inputs that helps with easing the routing pressure caused by a large block on the FPGA. Adding these DL-specific coarse-grained hard blocks to FPGAs increases their compute density and makes them even better hardware accelerators for DL applications, while still keeping the vast majority of the real estate on the FPGA programmable at fine-grain.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; **Neural networks**; • **Hardware →** *Arithmetic and datapath circuits*; **Hardware accelerators**; **Reconfigurable logic and FPGAs;**

Additional Key Words and Phrases: FPGA, neural networks, deep learning, machine learning, hardware acceleration, computer architecture, tensor slice

ACM Reference format:
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 (August 2022), 34 pages.
https://doi.org/10.1145/3529650

This research was supported in part by National Science Foundation (NSF) grant number 1763848. Any opinions, findings, conclusions or recommendations are those of the authors and not of the NSF.
Authors' addresses: A. Arora and L. K. John, Department of Electrical and Computer Engineering, The University of Texas at Austin, 2501 Speedway, Austin, TX, USA 78712; emails: aman.kbm@utexas.edu, ljohn@ece.utexas.edu; M. Ghosh, Department of Electrical Engineering, Indian Institute of Technology Kharagpur, Kharagpur, West Bengal, India 721302; email: moinakghosh2000@gmail.com; S. Mehta, Department of Electrical and Electronics Engineering, Bits Pilani K.K. Birla Goa Campus, Mormugao, Goa, India 403726; email: samidh99@gmail.com; V. Betz, Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G8; email: vaughn@ece.utoronto.ca.

1936-7406/2022/08-ART46 $15.00
https://doi.org/10.1145/3529650

1 INTRODUCTION

Deep Learning (DL) has become commonplace in today's world. Owing to the enormous computation requirements of DL workloads, many solutions have been deployed for accelerating them in hardware, ranging from ASICs to GPUs to FPGAs. FPGAs are very well suited for the rapidly changing world of DL. FPGAs provide massive parallelism, while being flexible and easily configurable, and also being fast and power-efficient.

At their heart, FPGAs comprise fine-grained programmable logic blocks (LBs), embedded memory structures (RAM blocks), and fixed-function math units (DSP slices). These generic building blocks make FPGAs flexible, but also limit the performance we can achieve with FPGAs for domain-specific applications like DL. That brings up a question: Can we improve the performance of FPGAs for DL workloads? Several techniques have been proposed or used by the industry and academia with this goal in mind. In this paper, we propose adding Tensor Slices to the FPGA, a block that is specialized for performing tensor operations like matrix-matrix and matrix-vector multiplication, which are common in DL workloads. This helps pack far more compute in the same area footprint and improves the performance of FPGAs for DL applications.

We architect and implement a Tensor Slice and compare the performance of an FPGA architecture with Tensor Slices (in addition to Logic Blocks, DSP Slices and RAM Blocks) with an FPGA architecture similar to state-of-the-art Intel's Agilex FPGAs. We convert a portion of the FPGA's area into Tensor Slices and observe a significant performance boost for DL benchmarks. We explore different percentages of the FPGA area spent on Tensor Slices.

FPGAs provide flexibility to meet the requirements of a broad range of applications. Adding Tensor Slices to an FPGA is focused on accelerating DL applications. It can be a concern that adding such slices may impact the generality of an FPGA, and hence may degrade the performance of non-DL applications by causing a higher routing wire length and longer critical paths. To that end, we also evaluate the impact of this proposal on non-DL applications. We also enhance the Tensor Slice so that it can be fractured into smaller math units like multipliers and MACs (multiply-and-accumulate) and used for non-DL workloads.

Our contributions in this paper are the following:

(1) We propose adding Tensor Slices to FPGAs, describing their architecture, design and implementation in detail.
(2) We quantify the impact of adding Tensor Slices on an FPGA for a variety of DL and non-DL benchmarks, by comparing their performance to a contemporary FPGA.
(3) We study the sensitivity of various metrics to the percentage of the FPGA area consumed by Tensor Slices.

An earlier and less detailed version of this work appeared in [5]. We extend that work significantly by using an improved architecture of the Tensor Slice that supports more precisions and operations, comparing the Tensor Slice with Intel AI Tensor Blocks [18] and Xilinx AI Engines [34], and enhancing the evaluation methodology by using DL benchmarks taken from Koios [4] and including more results.

The rest of this paper is organized as follows. The next section provides an overview of related work in the field of DL-specialized FPGAs. In Section 3, we present the architecture of the Tensor Slice. We explain the implementation details of the Tensor Slice, the FPGA architecture used for our experiments, and the benchmarks used for evaluation in Section 4. We compare the Tensor Slice with other similar DL-specialized blocks in Section 5. The results from the experiments we conducted are in Section 6.

2 RELATED WORK

In the last decade, many architectures have been proposed and deployed to meet the ever-increasing compute and memory demands of DL workloads. The Google TPU v1 [19] is an ASIC accelerator for DL applications, implementing a large 256×256 systolic array based matrix multiplication engine supporting int8 precision. NVIDIA Volta GV100 GPU [25] adds specialized cores called Tensor Cores to the Streaming Multiprocessors (SMs) on the GPU. Each Tensor Core provides a 2D processing array that performs the matrix multiplication operation on 4×4 matrices in fp16 precision. Different sizes of matrix units and more precisions have been added by Google and NVIDIA in later versions of these architectures.

Many FPGA based solutions exist as well. Microsoft's Brainwave [12], Intel's DLA [1], Xilinx's xDNN [32] are some examples. BrainWave is a soft NPU (Neural Processing Unit) with dense matrix-vector multiplication units at its heart implemented on Intel's Stratix 10 FPGAs. xDNN is an overlay processor, containing a systolic array based matrix multiplier, that can be implemented on Xilinx FPGAs. DLA uses a 1-D systolic processing element array to perform dot product operations commonly required in neural networks.

These accelerators use the programmable logic that exists on current FPGAs, such as LBs, DSPs and RAMs. They do not modify the architecture of the FPGA itself. There have been several innovations in changing the FPGA architecture as well. Intel's Agilex FPGAs [16] enable flexible integration of heterogeneous tiles using EMIB interface in a System-In-Package (SIP), an example of which is domain-specific accelerator tiles like Tensor Tiles [24]. Xilinx's Versal ACAP family [13] adds AI engines [33] on the same die as the programmable logic. These AI engines contain vector and scalar processors, with tightly integrated memory. Achronix's Speedster7t FPGAs [3] [2] have Machine Learning Processor (MLP) blocks in the FPGA fabric. An MLP is an array of up to 32 multipliers, followed by an adder tree, an accumulator, and a normalize block. Flex-Logix's nnMAX inference IP used in EFLX eFPGAs contains tiles in which 3×3 Convolutions of Stride 1 are accelerated by dedicated Winograd hardware [10, 11]. Intel's Stratix NX FPGAs [20] replace DSP Slices with blocks called AI Tensor Blocks [18]. These blocks have 30 MACs and 15× more int8 (8-bit fixed point) compute than a Stratix 10 DSP slice (7.5× compared to an Intel Agilex DSP slice). Native support for int4, fp16 and bfloat16 data types in DSP slices has also been added to recent FPGAs, e.g. Intel Agilex [16].

Optimizing FPGA architecture for DL has been researched in academia as well. Eldafrawy et al. [9] propose LB enhancements and add a shadow multiplier in LBs to increase MAC density in FPGAs improving DL performance. Boutros et al. [7] have explored enhancing DSP slices by efficiently supporting 9-bit and 4-bit multiplications. Rasoulinezhad et al. [26] suggest improvements to the DSP-DSP interconnect and also the inclusion of a register file in the DSP slice to improve data reuse. Nurvitadhi et al. [23] study integrating an ASIC chiplet, TensorRAM, with an FPGA in an SIP to enhance on-chip memory bandwidth. Arora et al. [6] propose adding hard matrix multipliers to FPGAs. These blocks support only one precision (fp16 or int8) and one operation (matrix-matrix multiplication).

We propose adding building blocks called Tensor Slices to the FPGAs. The Tensor Slice has 32× more int8 compute compared to Stratix 10 (16× compared to Agilex) DSP Slice. Tensor Slices support common precisions used in DL: int8, int16, fp16 and bf16. They support common operations like matrix-matrix multiplication, matrix-vector multiplication, element-wise matrix addition, subtraction and multiplication.

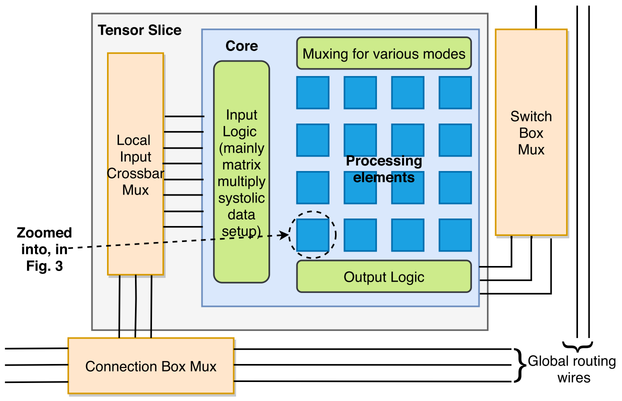


Fig. 1. High-level block diagram of the Tensor Slice.

3 ARCHITECTURE

3.1 Overview

A Tensor Slice is to Deep Learning, just like a DSP slice is to Digital Signal Processing. DSP Slices support the most common DSP operations like the MAC operation, along with additions and multiplications. Similarly, Tensor Slices support the most common machine learning operations like matrix-matrix multiplication and matrix-vector multiplication, along with element-wise matrix addition, subtraction and multiplication. The matrix-matrix and matrix-vector multiplication operations are pervasive in DL layers like fully-connected, convolution and recurrent layers. **Element-wise** (also referred to as **Eltwise**) matrix addition and subtraction is commonly found in layers like normalization, residual add and weight update. Eltwise matrix multiplication is used in layers like dropout. The Tensor Slice also has support for bias-preloading and tiling.

Figure 1 shows a logical block diagram of Tensor Slice. The slice interfaces with the FPGA interconnect through connection box (for inputs) and switch box (for outputs), similar to other blocks on modern FPGAs. The slice has a 50% sparsely populated local input crossbar, that makes the input pins of the slice swappable and hence increases the routability of the FPGA. The total number of inputs pins (including clock and reset) and output pins on the Tensor Slice are 310 and 298, respectively.

The core of the Tensor Slice is a 2D array of 16 processing elements (PEs) along with control logic. Each PE comprises of a multiplier and an adder which can act as an accumulator when a MAC operation is desired. The control logic comprises of input logic, output logic and muxing logic. The input logic sets the input data correctly (e.g. appropriately delay it for systolic computation) to be processed by the PEs. The output logic selects the output data appropriately from the PEs and shifts it out. The muxing logic selects between various modes of operation of the slice.

The Tensor Slice supports four precisions natively: 8-bit fixed-point (int8), 16-bit fixed-point (int16), IEEE half-precision (fp16), and Brain Floating Point (bf16) [15]. These are the most commonly used precisions in DL inference and training. In int8 mode, all multiplications happen in int8 but accumulations are done in 32-bit fixed-point (int32). In int16 mode, all multiplications happen in int16 but accumulations are done in 48-bit fixed-point (int48). In the fp16 and bf16 modes, all multiplications happen in fp16 and bf16, respectively, but accumulations are done in IEEE single precision (fp32).

There are two primary modes of operation of the Tensor Slice: Tensor mode and Individual PE mode. In the Tensor mode, the slice operates on matrix inputs, whereas in Individual PE mode, it

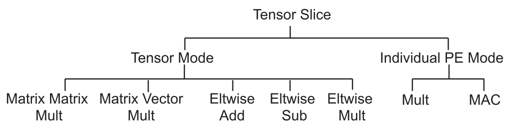


Fig. 2. Modes supported by the Tensor Slice.

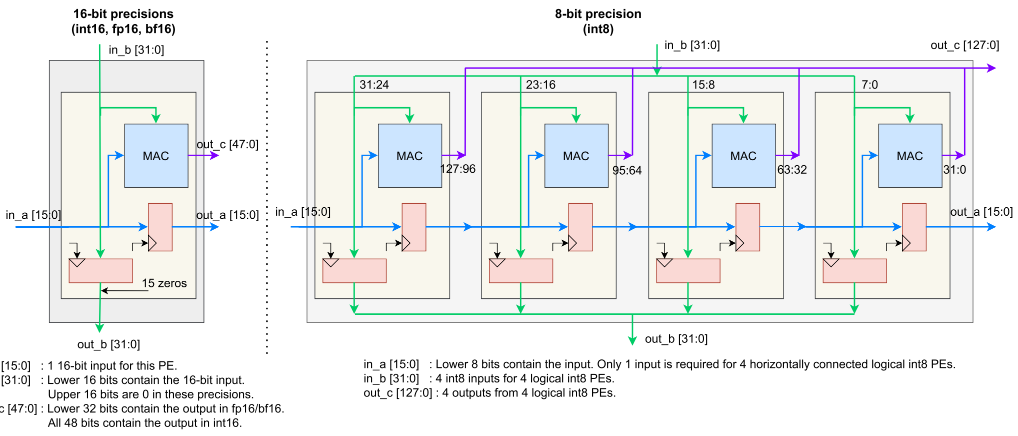

16-bit precisions (int16, fp16, bf16)
in_b [31:0]
MAC
out_c [47:0]
in_a [15:0]
out_a [15:0]
15 zeros
out_b [31:0]
8-bit precision (int8)
in_b [31:0]
out_c [127:0]
31:24
23:16
15:8
7:0
127:96
95:64
63:32
31:0
in_a [15:0]
out_a [15:0]
out_b [31:0]
in_a [15:0] : 1 16-bit input for this PE.
in_b [31:0] : Lower 16 bits contain the 16-bit input. Upper 16 bits are 0 in these precisions.
out_c [47:0] : Lower 32 bits contain the output in fp16/bf16. All 48 bits contain the output in int16.
in_a [15:0] : Lower 8 bits contain the input. Only 1 input is required for 4 horizontally connected logical int8 PEs.
in_b [31:0] : 4 int8 inputs for 4 logical int8 PEs.
out_c [127:0] : 4 outputs from 4 logical int8 PEs.

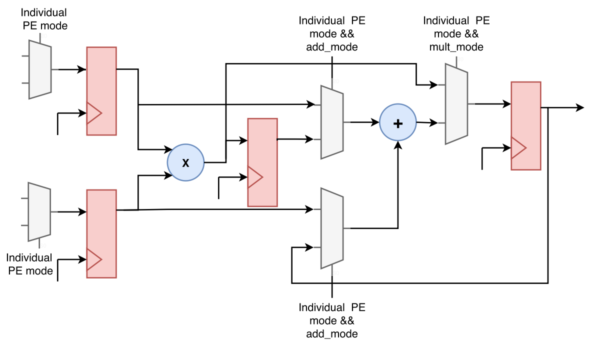
Fig. 3. A processing element (PE) in 16-bit and 8-bit precisions. There are 16 such PEs in the Tensor Slice.

operates on scalar inputs. There are five sub-modes of the Tensor mode: Matrix-Matrix Multiplication, Matrix-Vector Multiplication, Eltwise Addition, Eltwise Subtraction, and Eltwise Multiplication. There are two sub-modes of the Individual PE mode: Multiplier and MAC. All the modes and sub-modes supported by the Tensor Slice are shown in Figure 2. The mode of operation of the slice is dynamically selectable. That is, the mode bits can be changed during run-time without requiring reconfiguration of the FPGA.

3.2 Processing Element

For this section, we refer to each processing element (PE) in the Tensor Slice as a physical PE. We refer to the logic/circuitry required to process 1 matrix element as a logical or functional PE. There are 16 physical PEs in the slice. In 16-bit precision modes, the slice needs to process 16 matrix elements. So, there is a one-to-one correspondence between a physical PE in the slice and a logical PE required in 16-bit precision modes. For example, logical PE00 is physical PE00 of the slice, logical PE01 is physical PE01 of the slice, and so on upto PE33. However, in 8-bit precision mode, the slice processes 64 matrix elements, so it needs 64 logical PEs. Because of hardware sharing, each physical PE in the slice acts as 4 logical 8-bit PEs. So, physical PE00 in the slice maps to logical 8-bit PE00, PE01, PE02, PE03. Physical PE01 in the slice maps to logical 8-bit PE04, PE05, PE06, PE07. And so on. Figure 3 shows the diagram of one physical processing element (PE) configured for 8-bit precision operation (int8) as 4 logical PEs and for 16-bit precision operation (int16/fp16/bf16) as 1 logical PE.

Each PE consists of registers for shifting input data and a MAC. The MAC is shown in Figure 4. The figure also shows the multiplexing in the MAC required for the individual PE mode. Logically, the MAC consists of a multiplier and an adder. But to enable hardware sharing between the integer and floating point modes, the MAC contains multiple small-sized adders and multipliers which are combined to form larger adders and multipliers, along with multiplexing logic, floating-point


Individual PE mode
Individual PE mode
Individual PE mode && add_mode
Individual PE mode && mult_mode
Individual PE mode && add_mode
x
+
Fig. 4. Functional diagram of the MAC block which forms the core of a PE.

logic (aligning, normalizing, executing, rounding, etc.) and pipelining registers. There are four 8-bit multipliers and 16 8-bit adders in the MAC block.

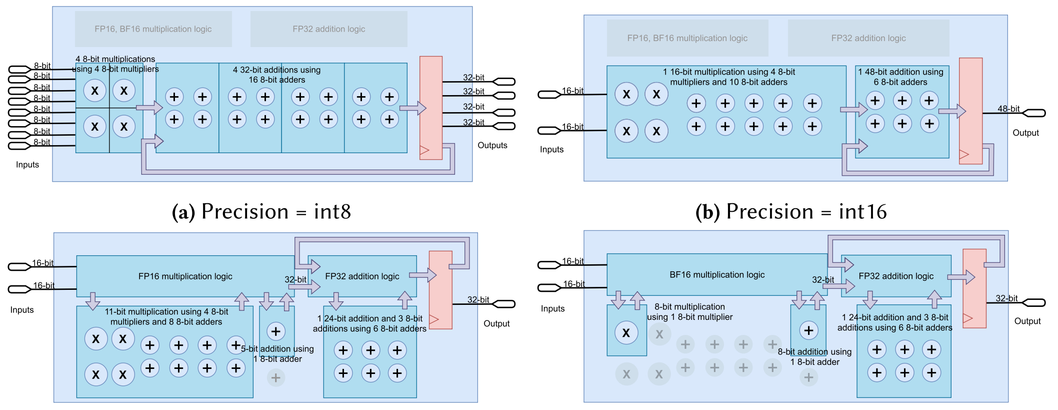
When operating in the int8 mode (Figure 5(a)), four int8 multiplications and four int32 additions are required. The four 8-bit multipliers are directly used and 4 int32 additions are performed by combining the 8-bit adders. When operating in the int16 mode (Figure 5(b)), one int16 multiplication and one int48 addition is required. The multiplication uses four 8-bit multipliers along with 10 8-bit adders to add the partial sums. The int48 addition is performed by combining 6 8-bit adders.

In floating point modes, the floating point logic reuses the 8-bit multipliers and 8-bit adders as required. In fp16 mode (Figure 5(c)), one fp16 multiplication and 1 fp32 addition are required. The fp16 multiplication logic needs to do an 11-bit multiplication (for mantissas), for which it uses the four 8-bit multipliers and 8 8-bit adders (to add partial sums). It also needs a 5-bit addition (for exponents), for which it uses one 8-bit adder. In bf16 mode (Figure 5(d)), one bf16 multiplication and one fp32 addition are required. The bf16 multiplication logic needs to do an 8-bit multiplication (for mantissas), for which it uses one 8-bit multiplier. It also needs a 8-bit addition (for exponents), for which it uses one 8-bit adder. For the fp32 addition (required by both fp16 and bf16 modes) uses the same hardware. In our implementation of the fp32 adder, it needed one 24-bit addition and 3 8-bit additions during its various stages. For this, it uses 6 8-bit adders. Some 8-bit adders stay unused in floating point modes.

3.3 Tensor Mode

The **I/O (Input/Output)** pins on the Tensor Slice in Tensor mode are shown in Table 1. The Tensor Mode is configured by setting the **`mode`** input to 0. When configured to use int8 precision (**`dtype`** = 00), the Tensor Slice acts on 8×8 matrix operands and generates an 8×8 matrix result. In int16 (**`dtype`** = 01), fp16 (**`dtype`** = 10) and bf16 (**`dtype`** = 11) precisions, the Tensor Slice acts on 4×4 matrix operands and generates a 4×4 matrix result. We observed that by doing this, we can utilize the I/O pins of the Tensor Slice fully in each mode. Also, there are ample opportunities to share hardware between 4×4 fp16/bf16/int16 and 8×8 int8 array of processing elements.

The Tensor Slice performs a tensor operation over multiple clock cycles. The input **`start`** is asserted to start the operation. The input matrices/vector would typically be stored in RAM blocks and some control logic implemented in soft logic would read the RAM blocks to feed the inputs to the slice. Alternatively, inputs may be generated from some upstream logic (e.g. hardware for the previous layer of a neural network) and fed directly into the slice without being stored in a RAM block. As the input matrices/vector are fed into the slice, control logic inside the slice orchestrates the data and applies the right data elements at the right time to specific PEs. When the output data is available in the PEs, it is sent out on **`c_data`** and **`flags`**. If out_ctrl is 0, the output data is


FP16, BF16 multiplication logic
FP32 addition logic
4 8-bit multiplications using 4 8-bit multipliers
4 32-bit additions using 16 8-bit adders
8-bit
32-bit
Inputs
Outputs
1 16-bit multiplication using 4 8-bit multipliers and 10 8-bit adders
1 48-bit addition using 6 8-bit adders
16-bit
48-bit
Output
FP16 multiplication logic
11-bit multiplication using 4 8-bit multipliers and 8 8-bit adders
5-bit addition using 1 8-bit adder
1 24-bit addition and 3 8-bit additions using 6 8-bit adders
BF16 multiplication logic
8-bit multiplication using 1 8-bit multiplier
8-bit addition using 1 8-bit adder
32-bit

(a) Precision = int8 **(b)** Precision = int16

(c) Precision = fp16 **(d)** Precision = bf16

Fig. 5. Sharing of arithmetic units (adders and multipliers) between different precisions of the MAC block. Multiplexers and pipeline registers not shown for clarity. Floating point logic refers to circuitry for alignment, normalization, rounding, etc.

Table 1. Inputs (I) and Outputs (O) of the Tensor Slice in Tensor Mode

I/O	Signal	Bits	I/O	Signal	Bits
I	clk	1	I	a_data_in	64
I	reset	1	I	b_data_in	64
I	mode	1	I	valid_mask_a_rows	8
I	accumulate	1	I	valid_mask_b_cols	8
I	preload	1	I	valid_mask_a_cols_b_rows	8
I	dtype	2	I	final_op_size	8
I	op	3	I	out_ctrl	1
I	start	1	O	b_data_out	64
I	x_loc	5	O	a_data_out	64
I	y_loc	5	O	c_data	160
I	a_data	64	O	c_data_available	1
I	b_data	64	O	flags	8
I	no_rounding	1	O	done	1

automatically shifted out cycle-by-cycle when it is ready, but the user can control when to shift it out by setting **out_ctrl** to 1. The output **c_data_available** is asserted when output data is valid on **c_data** and **flags**. **flags** contain the logical OR of the exception flags from the PEs in a column and are only valid for floating-point precisions. The output data can be stored in a RAM block, or directly fed to downstream logic (e.g. hardware for the next layer of a neural network) as it is generated by the slice. When the entire operation is done, the slice asserts the **done** signal.

Although the size of the matrix operations performed by the Tensor Slice are 4 × 4 and 8 × 8, the Tensor Slices can be chained to perform larger matrix operations. Section 3.3.4 provides details about this. Similarly, the Tensor Slice can support non-square inputs as well. For this purpose, there are validity masks for the inputs. This is done using **valid_mask_a_rows**, **valid_mask_a_cols_b_rows** and **valid_mask_b_cols** pins on the slice. For example, when multiplying a 6 × 4 matrix with a 4 × 7 matrix in int8 mode, the values of these inputs can be 8'b0011_1111, 8'b0000_1111 and 8'b0111_1111, respectively.

In Tensor mode, bias and tiling support can be enabled. For bias (controlled using **preload**), the Tensor Slice supports *pre-loading* the PEs with an input matrix which is effectively added to the result of the subsequent matrix operation. For tiling (controlled using **accumulate**), the Tensor Slice supports the choice of *not-resetting* the results in the PEs before starting another operation. This can be used in performing tiled or blocked matrix multiplications, where the partial sums need to be accumulated across tiles or blocks.

3.3.1 Matrix-Matrix Multiplication Mode. The matrix-matrix multiplication mode is enabled when **op** = 000. The matrix-matrix multiplication operation in the Tensor Slice is done systolically. Only the PEs along the left column and the top row of the 2D PE array receive external data. Other PEs receive data from neighboring PEs. The elements of first input matrix (matrix A) move from left to right and the elements of the second input matrix (matrix B) move from top to bottom. The result is calculated *during* the shifting process, and it *stays* in the respective PE until its computation is done. After that, the resulting matrix (matrix C) is shifted out left to right column-wise in a pipelined fashion. When the results are being shifted out, another tensor operation can be started on the Tensor Slice.

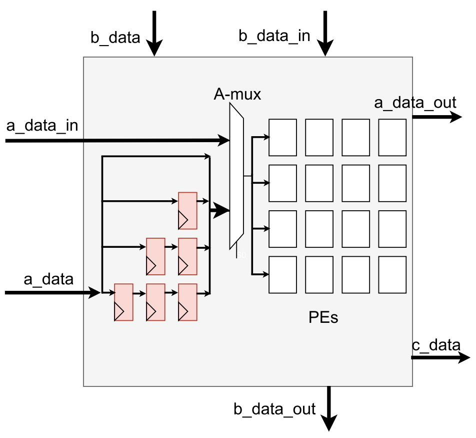
Elements of one operand matrix are applied column-wise on the input **a_data**. Elements of the second operand matrix are applied row-wise on the input **b_data**. In one cycle, 8 int8 elements or 4 int16/fp16/bf16 elements are applied on **a_data** and the same number of elements are applied to **b_data**. The output data is available on **c_data** and **flags**. In one cycle, results from one column of PEs are shifted out (See Section 3.3.5 for more details). Only 128 bits of c_data and 4 bits of flags are used in this mode.

Figure 6(a) shows the systolic setup of data from matrix A (left-to-right). See the path from **a_data** to the PEs through the flip-flops and A-mux. The muxing required for chaining (A-mux) that selects between **a_data** and **a_data_in** is discussed later in Section 3.3.4. Figure 6(b) shows the same logic but for data from matrix B (top-to-bottom). Figure 6(c) shows the movement of matrix A elements (in red) and matrix B elements (in yellow) through the PEs. Figure 6(d) shows the shift out of the results (i.e. data for matrix C).

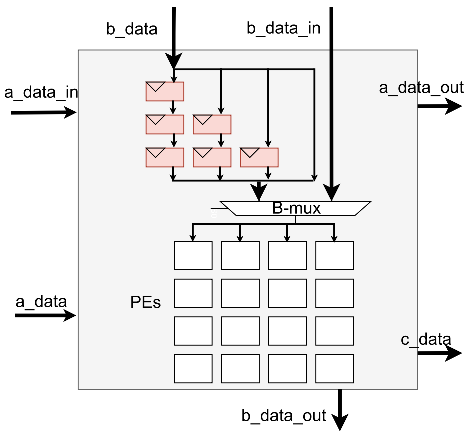
Matrix-matrix multiplication operation in the tensor mode is the most compute intensive operation done by the Tensor Slice. When using 16-bit precisions (int16, fp16, bf16), the slice performs 16 MAC operations in 1 cycle. So the math throughput of the slice is 16 MACs/clock. When using 8-bit precision (int8), the slice's math throughput is 64 MACs/clock. To keep the slice fed with data, it reads 8 16-bit elements every clock cycle in 16-bit precision modes and 16 int8 elements every clock cycle in int8 precision mode. So, the on-chip memory bandwidth requirement of the Tensor Slice is 16 bytes/clock.

3.3.2 Matrix-Vector Multiplication (matvec) Mode. The matrix-vector multiplication mode is enabled when **op** = 100. The matrix-vector multiplication operation in the Tensor Slice is also done systolically. The elements of the matrix move from left to right and the elements of the vector move from top to bottom. The result is calculated *during* the shifting process, and it *stays* in the respective PE until its computation is done. After that, the resulting vector is shifted out column-wise in 1 cycle.

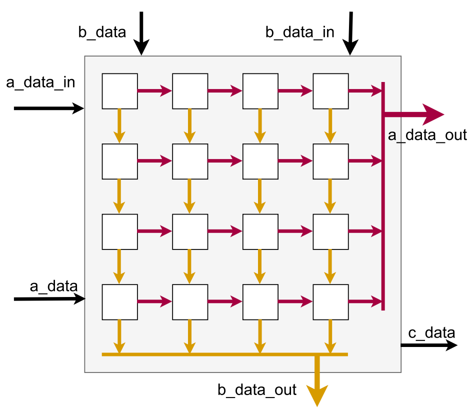
Elements of the input matrix are applied column-wise on the input **a_data**. Elements of the input vector are applied to **b_data**. Note that only 8 bits of **b_data** are used for int8 precision, and 16 bits of **b_data** are used for int16/fp16/bf16 precisions. Since there is only one column in a vector, this implies only the PEs in one column of the 2D PE array are utilized. We identify an opportunity to improve the utilization of PEs by observing that there are many I/O pins on the Tensor Slice that are required only for the matrix-matrix mode, but are not required in matrix-vector mode (and eltwise modes as well). We add multiplexers in front of the PEs in the third column and expose them



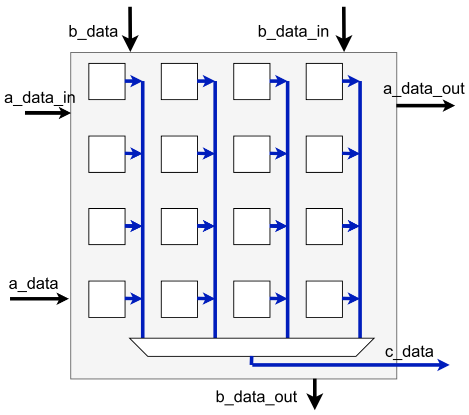
(a) Systolic data setup for Matrix A



(b) Systolic data setup for Matrix B



(c) Flow of input matrices within the Tensor Slice



(d) Flow of output matrix within the Tensor Slice

Fig. 6. Various aspects of operation of the Tensor Slice.

to already-existing unused I/O pins so that these PEs can also be loaded directly from the outside (instead of getting data from PEs to their left). Through this set of wires (called **`second_a_data`**), we can now feed another matrix in the matrix-vector mode. This is shown in Figure 7. This is a slight deviation from a pure systolic design, in which only the PEs on the periphery read/write data from outside. However, the overhead of adding this feature is low, and the utilization of the slice doubles in the matrix-vector mode. More multiplexers can be added to PEs in other columns and rows to further increase the utilization of the Tensor Slice in matrix-vector mode. However, this will require new I/O pins to be added to the Tensor Slice. I/O pins on a block in the FPGA fabric are costly in terms of area (larger local crossbar) and routing (more congestion). Adding multiplexers also increases the combinatorial delays of timing paths going through them. So, the cost-benefit tradeoff needs to be carefully studied before adding multiplexers to more columns and rows.

The second vector can be fed from the bits of b_data that are unused in this mode. With this enhancement, two independent matrix-vector products can be calculated at the same time in the slice. Some other unused I/Os can be used for validity masks and for reading out the output results. Here is the mapping of I/O pins used for the reading a second matrix and a second vector, and for outputting a second result in the Matrix-Vector Multiplication mode:

- **second_a_data** is mapped to a_data_in
- **second_validity_mask_a_rows** is mapped to validity_mask_b_cols
- **second_validity_mask_a_cols_b_rows** is mapped to b_data[23:16]
- **num_rows_matrix** is mapped to final_op_size
- **num_cols_matrix** is mapped to b_data[31:24]
- **second_b_data** is mapped to b_data[47:32]
- **second_c_data** is mapped to {c_data[159:128], b_data_out[63:48], b_data_out[31:16], a_data_out[63:0]}
- **second_flags** is mapped to flags[7:4]

The **num_rows_matrix** is used to specify the number of rows of the matrix, whereas the number of columns in the matrix (and hence the number of elements in the vector) is specified using **num_cols_matrix**. These are used inside the Tensor Slice to calculate the number of cycles elapsed to start shifting out the results and to assert the **done** signal.

In matrix-vector multiplication mode, when using 16-bit precisions, the slice performs eight MAC operations in one cycle (4 in column 1 and 4 in column 3). So the math throughput of the slice is 8 MACs/clock. When using 8-bit precision, the slice's math throughput is 16 MACs/clock. To keep the slice fed with data, it reads 10 16-bit elements every clock cycle in 16-bit precision modes, and hence the on-chip memory bandwidth requirement is 20 bytes/clock. Similarly, it reads 18 int8 elements every clock cycle in int8 precision mode, and hence the on-chip memory bandwidth requirement is 18 bytes/clock.

3.3.3 Eltwise Modes. Element-wise matrix operations are supported by the slice as well, and can be performed by selecting the right settings of the **op** pins (**op** = 001 => eltwise multiplication; **op** = 010 => eltwise addition; **op** = 011 => eltwise subtraction). For the eltwise operations, the elements of first matrix move left to right and the elements of the second matrix move from top to bottom. The result calculation happens *after* all inputs have reached their respective locations in the PE array. We observe that this method of moving data through the PEs in eltwise mode increases the number of cycles required for an eltwise operation.

The enhancement used for matrix-vector mode to increase the utilization of PEs can be extended to reduce the cycles required in eltwise mode by 2×. Instead of only feeding data into the 2D PE array from the left-column and top-row, additional PEs internal to the array can be fed, without adding any extra I/O cost. For matrix-vector multiplication mode, the third column was exposed on existing I/Os. In addition to exposing the third column, the third row is also exposed in eltwise mode. This enables loading of two columns of matrix A and two rows of matrix B at the same time, doubling the loading speed without adding any I/Os. The cost is a few multiplexers. I/Os unused in matrix-matrix multiplication mode are used for reading out the output results. This is also shown in Figure 7. The following list shows the mapping of I/O pins used for the reading a second matrix and a second vector, and for outputting a second result in Eltwise Modes:

- **second_a_data** is mapped to a_data_in
- **second_b_data** is mapped to b_data_in
- **second_c_data** is mapped to a_data_out[63:0]
- **second_flags** is mapped to flags[7:4]

3.3.4 Chaining. Multiple Tensor Slices can be chained to perform operations on larger matrices. This is useful in matrix-matrix and matrix-vector multiplication operations. Figure 8 shows a logical view of four Tensor Slices chained in x and y directions to perform a larger matrix-matrix multiplication operation (e.g. an 8 × 8 matrix multiplied with an 8 × 8 matrix using four slices in

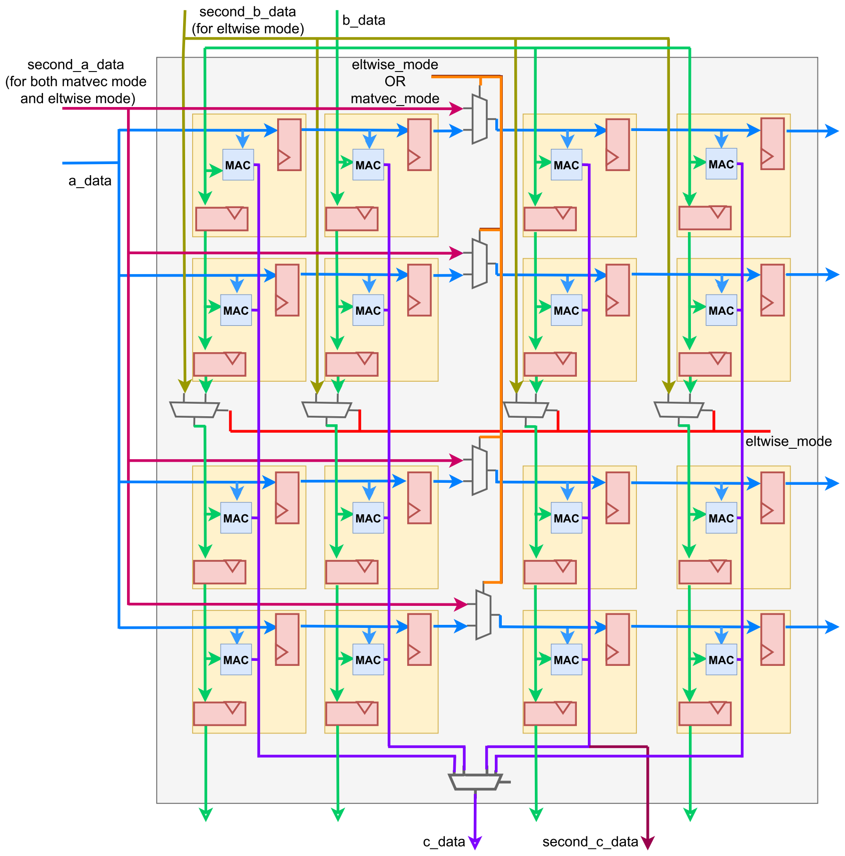


Fig. 7. Exposing internal PEs to increase the utilization in matrix-vector multiplication mode and reduce the number of cycles in eltwise modes. Note that no new I/Os are used; instead I/Os that would have been unused in these modes are repurposed.

fp16 mode). Signals **a_data_in** and **a_data_out** are used to chain the inputs from matrix A along the x direction. The signals **b_data_in** and **b_data_out** are used to chain the inputs from matrix B along the y direction. Only the Tensor Slices at the periphery are fed inputs. Inputs flow through the slices through the chains. The **c_data** signal contains the output of the Tensor Slice. It can be chained with the output from neighboring Tensor Slices using soft logic or directly consumed from each Tensor Slice block, depending the requirements of the user's design.

Note that the figure shows a logical connectivity of the slices in the x (horizontal) and y (vertical) directions. Physically, these slices can be anywhere on the FPGA. For example, four Tensor Slices in one grid column of the FPGA could be connected to perform a larger matrix operation. The inputs **x_loc** and **y_loc** are used to specify the logical location to the slices. Note that **x_loc** and **y_loc** do not determine or are related to the physical location of a slice in the FPGA grid. These signals are decoded internally to select the correct input port(s) whose data should feed the PEs. For example, the top-left slice in the logical grid of slices has **{x_loc,y_loc}=00**, implying that this slice should use the input data received at **a_data** and **b_data** ports. The bottom-right slice in the logical grid of slices has **{x_loc, y_loc}=11**, implying that this slice should use the inputs received at **a_data_in** and **b_data_in** from its logical neighbors to the left and top, respectively. Not only do different slices in a logical grid receive data from different ports, they get data at

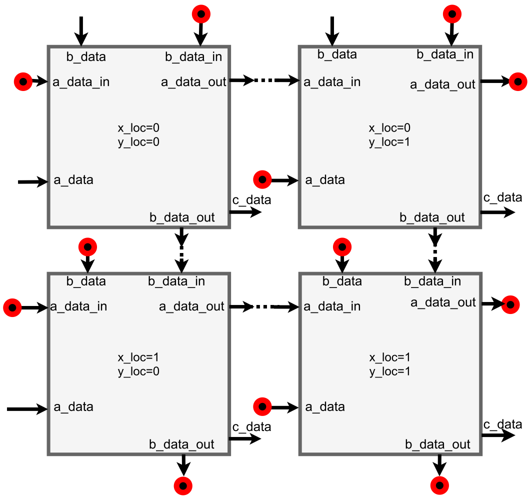


Fig. 8. Multiple Tensor Slices can be chained together to perform larger matrix-matrix multiplications. Here, four slices are shown to be chained together in x & y direction (logically).

different times as well. For example, the inputs going into slice with **{x_loc, y_loc}=11** are delayed with respect the inputs going in to the slice with **{x_loc, y_loc}=00**. **x_loc** and **y_loc** are also used in the control logic in the slice to sample the incoming data at the appropriate time.

The input **final_op_size** is used to specify the overall size of the matrix operation being performed. In the case of the example shown in Figure 8, assuming int16 operation, the **final_op_size** will be set to 8, because four slices are connected together and each slice performs a 4×4 matrix operation. This signal is used in the control logic in the slice to determine when the computation is finished and when to start shifting out the result.

Consider a matrix-matrix multiplication where an MxK matrix is multiplied with a KxN matrix. For large values of M, the Tensor Slices are chained in the y (logically vertical) direction. For large values of N, the Tensor Slices are chained in the x (logically horizontal) direction. For large values of K, instead of chaining, typically, a longer number of cycles is used to accumulate the results. In other words, M and N dimensions are handled by using more hardware (more "space"), whereas K dimension is handled by using more cycles (more "time"). An advantage of mapping the K dimension on "time" is that the extended precision intermediate results do not need to move. The same concept applies to a matrix-vector multiplication, except that there is no requirement of chaining in the x (logically horizontal) direction. It only makes sense to chain Tensor Slices in the y (logically vertical) direction.

3.3.5 Rounding. As mentioned above, accumulations in the Tensor Slice are done at a higher precision, compared to the multiplication. In other words, the results have higher precision compared to the operands. When **no_rounding** is set to 1, the outputs are shifted out in the higher precision without being rounded to the input precision. But when **no_rounding** is set to 0 by the user, the outputs are rounded to input precision before being shifted out. Convergent rounding or "round half to even" rounding [31] is used to round the results. When rounded results are shifted out, it can take less number of cycles depending on the precision. For example, in the matrix-matrix multiplication int8 mode, if rounding is disabled, the output from 8 PEs (8 PEs = 1 column of PEs for int8 precision) is 8 * 32 = 256 bits. The **c_data** signal is 128 bits. So, it takes 16 cycles to shift out the data of all 8 columns. However, if rounding is enabled, the output from 8 PEs is 8 * 8 = 64 bits.

So, it takes 8 cycles to shift out data of all 8 columns. In matrix-matrix multiplication fp16 mode, if rounding is disabled, the output from 4 PEs (4 PEs = 1 column of PEs for fp16 precision) is 4 * 32 = 128 bits. So, it takes 4 cycles to shift out the data of all 4 columns. If rounding is enabled, the output from 4 PEs is 4 * 16 = 64 bits. So, in this case also, it takes 4 cycles to shift out data of all 4 columns.

3.4 Individual PE Mode

When the **mode** input pin is set to 1, the Tensor Slice changes to Individual PE mode. The main goal of providing this mode is to reduce the impact of adding Tensor Slices to an FPGA on non-DL applications and to improve utilization. In this mode, the slice is fractured such that inputs and outputs of individual PEs are exposed to the pins of the slice, enabling the PEs to be used like mini-DSP slices. Each PE can be separately and dynamically configured in two sub-modes: Multiplier or MAC. Furthermore, all the four precisions (int8, int16, fp16 and bf16) are available and can be dynamically selected. In int8 mode, each PE can be configured to be used as 2 8-bit multipliers or 1 8-bit MAC with 32-bit accumulation. In int16 mode, each PE can be configured to be used as 1 16-bit multiplier. In fp16 and bf16 modes, each PE can be configured to be used as 1 16-bit multiplier or 1 16-bit adder or 1 16-bit MAC with fp32 accumulation. Note that because of the large delay to access the PEs in the Tensor Slice (because of the local input crossbar), using the Individual PE mode will not be performant compared to, for example, a DSP slice based multiplication or MAC.

There is a limitation of this mode. The number of inputs and outputs on the slice (also called the I/O footprint of the slice) is governed by the Tensor mode (310 inputs, including clock and reset, and 298 outputs). Based on that, 8 PEs out of the 16 PEs can be exposed. We could add additional inputs and outputs to the slice to accommodate for exposing all 16 PEs in individual PE mode, but that would mean worsening the I/O footprint of an already large slice. Increasing the number of I/Os can lead to more routing congestion and higher channel width requirement and also a larger area of the Tensor Slice.

The inputs and outputs of an exposed PE are:

- **direct_in_a[15:0]**
- **direct_in_b[15:0]**
- **direct_mode** (Multiply or MAC)
- **direct_dtype[1:0]** (int8, int16, fp16 or bf16)
- **direct_out[31:0]**
- **direct_flags[3:0]** (exception flags for floating point mode)

Each exposed PE's inputs and outputs are mapped onto the top-level inputs and outputs of the slice (shown in Table 1). The mapping of all inputs and outputs to various PEs is not significant for this paper, but here's an example of the pin mapping for exposed PE #1:

- **direct_in_a[15:0]** is mapped to {valid_mask_b_cols, final_op_size}
- **direct_in_b[15:0]** is mapped to a_data[31:16]
- **direct_mode[1:0]** is mapped to x_loc[3:2]
- **direct_dtype** is mapped to accumulate
- **direct_out[31:0]** is mapped to c_data[31:0]
- **direct_flags[3:0]** is mapped to b_data_out[7:4]

4 EVALUATION METHODOLOGY

The goal of evaluating the Tensor Slice is to compare the performance of an FPGA with Tensor Slices, LBs, DSPs, and RAMs on it, with an FPGA with only the traditional building blocks (LBs, RAMs, and DSPs). We use an Intel Agilex-like FPGA architecture as our baseline. There are differences between our baseline architecture and Intel Agilex (e.g. we do not model HyperFlex), but for the purposes of this evaluation, we believe that as long as the baseline and the proposed FPGA architectures only differ in presence/absence of Tensor Slices, the results will hold. Also, in our evaluation, we use 22nm technology node, but Intel Agilex devices are 10nm.

4.1 Tools Used

To evaluate and compare FPGA architectures, we use the Verilog-to-Routing (VTR) tool flow [22]. VTR takes two inputs. The first input is an architecture description file containing information about an FPGA's building blocks and interconnect resources. The second input is a Verilog design. VTR synthesizes and implements the design on the given FPGA architecture and generates resource usage, area and timing reports.

To obtain the area and delay values for the various components of an FPGA (to enter them in the FPGA architecture description file for VTR), we use COFFE [35]. COFFE is a transistor sizing and modeling tool. The inputs to COFFE are the properties of the FPGA architecture (e.g. N, K, Fcin, Fs, etc.). It performs SPICE simulations to iteratively optimize the transistor sizing and generates areas and delays in various subcircuits in the FPGA tile. COFFE also supports a hybrid flow in which the core of blocks like DSP Slices or Tensor Slices is implemented using a standard cell flow and the interface to the interconnect (local crossbar, switch box, connection box, etc.) is implemented in full custom using SPICE. The standard cell flow uses Synopsys Design Compiler for synthesis, Cadence Encounter for placement & routing, and Synopsys Primetime for timing analysis. Our SPICE simulations use 22nm libraries from Predictive Technology Model [28]. Our standard cell library was the 45nm GPDK library from Cadence. We used scaling factors from Stillmaker et al. [27] to scale down from 45nm to 22nm. When running COFFE, we used a cost factor of $area*delay^2$ as it reflects the greater emphasis on delay compared to area, which is typical of high-performance FPGAs like Agilex.

4.2 DSP Slice Implementation

We create a DSP slice that closely matches the DSP slice from Intel Agilex DSP user guide [17]. It supports all major modes and all precisions - 9×9, 18×19, 27×27, fp16, bf16, fp32. The slice also supports input chaining (scanin-scanout) and output chaining (chainin-chainout). Our Agilex-like DSP slice does not include some features that are present in the Agilex DSP slice. For example, we have limited support for internal coefficients and constants, no double accumulation support, limited support for bypassing pipeline registers, and no enable signals for registers. To sanity check our implementation, we first implemented a fixed-point-only slice (Intel Arria) and compared the area and delay numbers to those in [7]. Our numbers were very similar after scaling for technology nodes. For floating-point units, we use the architecture described in [8] (hard multiplier and adder based, not the soft logic based, not the iterative design; also, we use a different pipelining scheme). The round to nearest tie-breaks-to-even (RNE) rounding scheme is used. Support for exceptions is present as well.

For our Agilex-like DSP slice, the critical path delay came out to be 2.93ns (341 MHz) in floating point mode and 2.33ns (429 MHz) in fixed point mode. The delay of the 50% sparsely populated local crossbar was 0.33ns. Our DSP slice has 130 non-dedicated inputs and 74 non-dedicated outputs. It spans 4 rows in the FPGA grid (1 LB spans 1 row) and the DSP slice column is 1.6× wide (compared to that of a LB). Table 2 shows the breakdown of the DSP slice area obtained from COFFE. We observe that about 40% of the area of the slice is routing, whereas 60% is the core. Table 3 shows the post-synthesis area of the DSP Slice as we added more precisions to it.

Table 2. Breakdown of the DSP Slice Area (Post P&R)

Component	**Area** (um^2)
Standard-cell core	7701
Local input crossbar	1480
Dedicated output routing	26
Switch box (4)	2736
Connection box	652
Total	**12597**

Table 3. Overhead of Supporting More Precisions in the DSP Slice (Post-synthesis Area Ratios)

DSP Slice variant	**Ratio**
Fixed point 18×19 and 27×27 (similar to Intel Arria)	1
Add fp32 support (similar to Stratix 10)	1.26
Add support for fp16, bf16 and fixed point 9 bit (similar to Agilex)	1.51

Table 4. Breakdown of the Tensor Slice Area (Post P&R)

Component	**Area** (um^2)
Standard-cell core	45404
Local input crossbar	2771
Dedicated output routing	0
Switch box (8)	5473
Connection box	1570
Total	**55219**

4.3 Tensor Slice Implementation

We implement a Tensor Slice using the architecture described in Section 3. The Tensor Slice uses the same core fixed-point and floating-point adders and multipliers as the ones used in the DSP slice. The number of inputs in the Tensor Slice is much larger than that of the DSP Slice, so the local input crossbar delay is higher (0.765ns). The critical path delay of the Tensor Slice is 3.31ns (302 MHz) in floating point mode and 2.56ns (391 MHz) in fixed point mode. The Tensor Slice has 308 non-dedicated inputs and 298 non-dedicated outputs. It spans 8 rows in the FPGA grid and the Tensor slice column is 3.5× wide (compared to that of a LB).

Tables 4 and 5 show the breakdown of the Tensor Slice area obtained from COFFE. We observe that about 18% of the area of the slice is routing, whereas 82% is the core. Within the core, ~90% of the area is consumed by the PE array. Inside each PE, the adder takes ~44% area and the multiplier takes ~30% area. The adder takes more area than the multiplier primarily because of the presence of fp32 adder. Table 6 shows the post-synthesis area of the Tensor Slice as we added more modes to it.

Table 5. Area Distribution of the Various Components of the Tensor Slice Core (Left) and the Processing Element (Right)

Component	Area (%)
Input logic	3.14
Output logic	6.25
2D PE array	90.61
Total	**100**

Component	Area (%)
Adder	44.1
Multiplier	29.7
Rest	26.1
Total	**100**

Table 6. Overhead of Adding More Functionality/Modes/Precisions in the Tensor Slice (Post-synthesis Area Ratios)

Tensor Slice variant	Ratio
8×8 int8 matrix mult only	1
4×4 fp16 matrix mult only	0.89
4×4 fp16 and 8×8 int8 matrix mult	1.34
Add individual PE modes	1.59
Add bf16 and int16 modes	1.86
Add element-wise and matrix-vector multiplication modes	2.10

4.4 Baseline vs. Proposed FPGAs

The routing and tile parameters of the FPGA architecture used for the baseline and proposed FPGAs are shown in Table 7. These are based on modern Intel FPGAs. Blocks on the FPGAs (LBs, DSPs, etc.) are arranged in columns. There are no sectors or super-logic-regions. I/O pads are arranged along the perimeter of the FPGA. Unidirectional routing with wire segments of length 4 (260 out of 300 wires) and length 16 (40 out of 300 wires) are used. The switch blocks use a custom switching pattern based on the Stratix-IV-like architecture used in the Titan flow [21]. Each logic block (LB) contains 10 basic logic elements (BLEs). Each BLE has a 6-input LUT which can be fractured into two 5-input LUTs. The BLE also has two flip-flops and 2 bits of arithmetic with dedicated carry chains between LBs. RAM blocks have a capacity of 20 Kilobits and have registered inputs and outputs. True and simple dual port modes with varied heights and widths (512×40, 1024×20, 2048×10) are supported. DSP Slice supports multiple precisions - 9×9, 18×19, 27×27, fp16, bf16, fp32 and has almost all the modes and features present in Intel Agilex DSP Slice. We use the FPGA architecture file provided with the Koios benchmarks [4] because it matches these features. We modify it to update the properties of our DSP Slice (Section 4.2) to create the baseline FPGA architecture. We then add the Tensor Slice block (Section 4.3) to create the proposed FPGA architecture.

From Intel Agilex's product table, we pick the product with the most compute intensive resource mix: AGF 027 (91280 LBs, 8528 DSP slices and 13272 RAM blocks). Based on the areas of each block on the FPGA obtained from COFFE, we calculate the total area of the FPGA consumed by each type of block. For our baseline architecture, we use the exact same resource mix as Intel Agilex AGF 027 in terms of percentage of the area and percentage of the count of various blocks on the FPGA. The absolute size of the FPGA is smaller than Agilex to ensure speedy simulations and also to ensure high utilization of the FPGA for our benchmarks for realistic results. Table 8 shows the total count of each block type, the percentage of total area spent on each block type and the percentage count of each block type for Intel Agilex AGF 027 and our baseline FPGA. We then create six

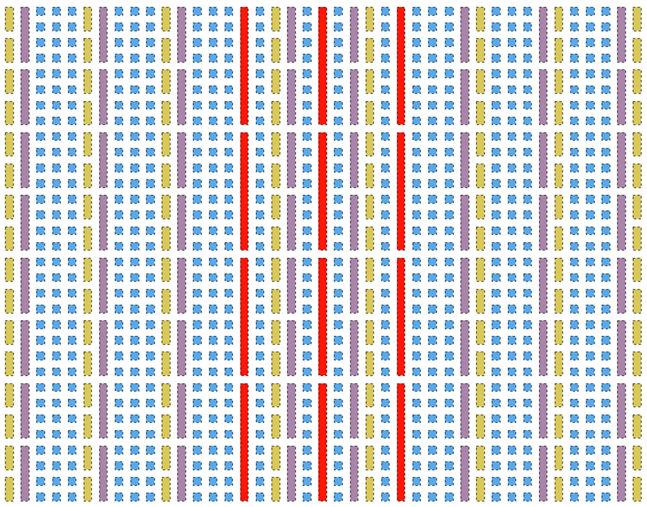


Fig. 9. A zoomed-in version of the Prop_5pct FPGA architecture obtained from VTR. Blue: Logic Block, Yellow: RAM Block, Purple: DSP Slice, Red: Tensor Slice. This is not a physical layout; different column types have different widths in the actual layout.

variations of the proposed FPGA architecture by spending 5%, 10%, 15%, 20%, 25%, 30% area of the FPGA on Tensor Slices, respectively. These architectures are referred to as "Prop_Xpct", with X taking a value from {5,10,15,20,25,30}. The Table 9 shows the resource mix of these FPGA architectures. The main goal of creating multiple variations of the proposed FPGA is to study the sensitivity of various metrics for non-DL benchmarks to increasing the area consumed by Tensor Slices on an FPGA.

The grid dimensions of all the architectures used in our experiments were around 170×80. The number of columns containing Tensor Slices were 3, 6, 9, 12, 15, and 18 in Prop_5pct, Prop_10pct, Prop_15pct, Prop_20pct, Prop_25pct, and Prop_30pct architectures, respectively. We placed the Tensor Slice columns in the middle of the FPGA to keep the layout symmetric and to ensure shorter paths between neighboring Tensor Slices. This is an important aspect of the proposed architecture and is governed by the reduction in frequency observed when Tensor Slice columns were placed far apart in the FPGA. Figure 9 shows a part of the Prop_5pct FPGA showing the three Tensor Slice columns in this architecture.

4.5 Benchmarks

For benchmarking our FPGA architecture, we use a set of DL and non-DL designs. Table 10 contains a list of all the benchmarks we used along with a brief description of the nature of each workload.

4.5.1 DL Benchmarks. For DL benchmarks, we use several designs from the Koios benchmark suite [4]. These designs cover various sub-applications within DL like Multi-Level Perceptrons, Convolutional Neural Networks, Recurrent Neural Networks, etc. We added design variations to cover multiple precisions - int8, int16, bf16, and fp16. When evaluating DL benchmarks on the proposed FPGA(s), Tensor Slices were manually instantiated in Verilog, because the synthesis tool cannot automatically infer them. The process of conversion of existing benchmarks that use DSP slices to new benchmarks that use Tensor Slices involved:

- Identifying occurrences of matrix-matrix multiplication, matrix-vector multiplication and elementwise operations in the Verilog code of the benchmarks

Table 7. Routing and Tile Architecture Parameters of the Baseline and Proposed FPGA Architectures

Parameter	Value	Definition
N	10	Number of BLEs per cluster
W	300	Channel width
L	4,16	Wire segment lengths
I	60	Number of cluster inputs
O	40	Number of cluster outputs
K	6	LUT size
Fs	3	Switch block flexibility
Fcin	0.15	Cluster input flexibility
Fcout	0.1	Cluster output flexibility
Fclocal	0.5	Local input crossbar population

Table 8. Resource Mix in Agilex AGF047 and Our Baseline FPGA Architecture

		AG047			Baseline		
Block	Relative area	# blocks	% area	% count	# blocks	% area	% count
Logic Block	1	91280	48.52	80.72	8480	45.19	80.92
DSP Slice	6.5	8528	29.49	7.54	800	27.37	7.63
RAM Block	3.12	13272	21.99	11.74	1200	27.44	11.45
Tensor Slice	28.5	0	0	0	0	0	0

Table 9. Resource Mix in Various Variations of the Proposed FPGA Architecture

	Prop_5pct		Prop_10pct		Prop_15pct		Prop_20pct		Prop_25pct		Prop_30pct	
Block	#	%	#	%	#	%	#	%	#	%	#	%
Logic Block	8000	45.59	7920	45.11	7760	44.37	7680	43.89	7600	43.4	7120	40.69
DSP Slice	780	28.92	720	26.68	660	24.55	600	22.3	540	20.06	500	18.59
RAM Block	1160	20.61	1040	18.47	920	16.4	800	14.25	680	12.11	640	11.4
Tensor Slice	30	4.88	60	9.74	90	14.67	120	19.55	150	24.43	180	29.33

Prop_Xpct refers to a variation of the proposed FPGA where approximately X% of the FPGA area is consumed by Tensor Slices. # denotes the number of blocks of a type. % denotes the percentage of area consumed by that block type.

- Replacing that code with Tensor Slice instances i.e. instantiating the module called `tensor_slice` with the same port list as defined in the VTR architecture file
- Designing appropriate control logic modules (Finite State Machines) that orchestrate the data movement in and out of the slice (read data stored in RAMs, feed it to the slice at the right time, write outputs to RAMs, etc.)
- Making connections between the new control logic, the Tensor Slice instances and the rest of the existing code
- Verifying the operation of the new design with simulation and comparing the results with the original unmodified benchmark

In two benchmarks (`lstm` and `attention`), the process of mapping to Tensor Slices involved some changes to the original architecture of the benchmark. We modified the designs keeping them as close to the original architecture as possible - in terms of clock cycles spent, data elements consumed per cycle, etc.

Table 10. Non-DL and DL Benchmarks Used for Evaluation Shown in Increasing Order of Number of Netlist Primitives

Benchmark	Netlist primitives	Nature of the workload
Non-DL benchmarks		
`or1200`	4305	Soft processor. Some DSP and RAM usage.
`blob_merge`	11527	Image processing. No DSP or RAM usage. Only soft logic.
`arm_core`	18156	Soft processor. No multiplications in this design, but some RAM usage.
`stereovision1`	29075	Computer vision. Some fixed point multiplications and no RAM usage.
`stereovision0`	31090	Computer vision. No multiplications in this design and no RAM usage.
`LU8PEEng`	39042	Math. Has floating point operations and some RAM usage.
`bgm`	42293	Finance. Has floating point operations and no RAM usage.
`stereovision2`	68683	Computer vision. Lots of fixed point multiplications and no RAM usage.
`LU32PEEng`	128132	Math. Has floating point operations and extensive RAM usage.
`mcml`	178845	Medical physics. Large design with multiple fixed point multiplications.
DL benchmarks		
`eltadd`	10778	A design that adds two 6×14 int8 matrices elementwise. Inputs read from RAMs and output stored into RAMs.
`fcl.int`	19426	Fully connected layer using int8 precision. Has num_features = 15, batch_size = 16 and num_outputs = 14. APB based control and configuration logic. RAMs store inputs and outputs.
`conv.fp`	22603	Convolution using 8×8 fp16 input image with 3 channels, padding=1, stride=1, filter size = 3×3 and batch size=2.
`eltmul`	23437	A design that multiplies two 24×8 fp16 matrices stored in RAMs, elementwise. Inputs and outputs stored into RAMs.
`tpuld.16`	29722	A design similar to Google's TPU v1 [19] with a 16×16 systolic array and normalize, pool and activation units. RAMs store activation and weights. Precision = int8.
`conv.int`	32299	Same convolution design as above, but with int16 precision.
`attention`	51858	Self-attention module in Transformers [29]. Involves matrix vector multiplication, elementwise multiplication and softmax (precison = int16).
`fcl.bf`	64758	Fully connected layer using bf16 precision with AXI programming interface. No RAM usage. 20×20 activation matrix, 20×20 weight matrix and 20×20 output matrix.
`tpuld.32`	96497	Same TPU design as above, but with a 32×32 systolic array.
`lstm`	340460	An LSTM layer [14] involving several matrix vector multiplications, elementwise operations and activations (precision = int16).

The netlist primitives are from when the benchmark is implemented on the baseline FPGA with DSP slices.

4.5.2 Non-DL Benchmarks. For non-DL benchmarks, we use the VTR benchmark suite [22]. These designs cover several domains like computer vision, medical physics, math, finance, etc. These include designs with/without floating point operations, heavy/low DSP usage, and heavy/low/no RAM usage. We used the 10 largest designs in the benchmark suite (based on number of netlist primitives), so that the utilization of the FPGAs (baseline and proposed) was fairly high to ensure realistic results.

5 DISCUSSION

5.1 Benefits and Limitations

In this section, we present the benefits of adding Tensor Slices to FPGAs, qualitatively. Quantitative results are shown in Section 6.

- Adding Tensor Slices to an FPGA increases the compute density of the FPGA. The area consumed by DL designs is smaller when Tensor Slices are used. This implies larger DL designs can now fit the same-sized FPGA chip.
- Designs using Tensor Slices can achieve faster frequencies compared to those using DSP Slices, because of the reduced dependence on routing/interconnect. This means DL applications can run faster on FPGAs with Tensor Slices.
- Using Tensor Slices leads to a reduction in routing wirelength required to implement a DL design. This implies a reduction in power consumption as well.
- Tensor Slices are large coarse-grained blocks. A design using Tensor Slices can go through the FPGA tool chain (synthesis, packing, placement, routing) faster. This means shorter turn around time for debug iterations.

Overall, Tensor Slices lead to better out-of-the-box performance for DL applications. This can lower the barrier of adoption for FPGAs for DL acceleration. Note that, currently, to use Tensor Slices, a designer has to manually instantiate a Tensor Slice block in the RTL and connect it. This is commonly how DSP Slices are used as well, other than for simple use cases like multiplication and MAC. In the future, synthesis tools can be improved to infer some code patterns in RTL and map the computation to a Tensor Slice instead. This will prevent users from having to manually instantiate Tensor Slices. Pragmas can be used to aid synthesis tools in mapping code to Tensor Slices. High-Level Synthesis (HLS) tools can generate RTL that instantiates Tensor Slice blocks.

There are some limitations of adding Tensor Slices to FPGAs as well. First, adding Tensor Slices makes an FPGA more heterogeneous (any new type of block added to the FPGA fabric adds to heterogeneity). This increases the complexity in CAD tool algorithms, especially at the implementation stage (pack, place, route). However, this cost is typically transparent to end-users of the FPGA. Secondly, adding Tensor Slices makes an FPGA less generic or flexible compared to a typical FPGA. If a Tensor Slice is not required or can not be used by an application (even in individual PE mode), the Tensor Slices will remain un-utilized on the FPGA. But with the abundance of DL applications, DL-specialized FPGA families containing Tensor Slices can be created. And non-DL applications can continue to use non-DL-specialized FPGAs.

5.2 Comparison with DSP Slice

Table 11 compares I/O and area related properties of the DSP Slice with the Tensor Slice. The Tensor Slice is about 4.3× in area compared to our implementation of an Agilex-like DSP Slice, but has only 1.5× the number of I/O pins. This means that the Tensor Slice has a low pin density or that the Tensor Slice has a high core-to-pin ratio. The Tensor Slice also has about 3× the logic-to-routing ratio, compared to the DSP Slice. The routing area includes the local crossbar, the switch boxes and the connection boxes. This implies significantly lower routing overhead compared to a DSP slice.

Table 12 compares throughput related properties of the DSP Slice with the Tensor Slice. The Tensor Slice has over 14× int8 throughput and over 7× throughput for 16-bit precisions, compared

Table 11. Comparing I/O and Area Related Metrics for a Tensor Slice with a DSP Slice (Numbers Based on 22nm Tech Node)

Metric	DSP Slice	Tensor Slice	Ratio
Area (um^2)	12597	55218	4.3
Number of I/Os	386	608	1.5
Pin density (pins/um^2)	0.03	0.01	0.3
Logic to routing area ratio	1.5	4.5	3

Table 12. Comparing Compute Throughput Related Metrics of a Tensor Slice with a DSP Slice (Numbers Based on 22nm Tech Node)

Precision	Metric	DSP Slice	Tensor Slice	Ratio
INT8	Number of MACs	4	64	16
	Freq (MHz)	429	391	0.9
	Throughput (GigaMACs/sec)	1.7	25.0	14.6
	Throughput/area (GigaMACs/sec/mm^2)	136.2	453.2	3.3
INT16	Number of MACs	2	16	8
	Freq (MHz)	429	391	0.9
	Throughput (GigaMACs/sec)	0.8	6.2	7.3
	Throughput/area (GigaMACs/sec/mm^2)	68.1	113.3	1.6
FP16/BF16	Number of MACs	2	16	8
	Freq (MHz)	341	302	0.9
	Throughput (GigaMACs/sec)	0.7	4.8	7.0
	Throughput/area (GigaMACs/sec/mm^2)	54.1	87.5	1.6

to an Agilex DSP slice. Note that because of quantization/fragmentation, the Tensor Slice can suffer from under-utilization (reduced effective throughput) in cases where the problem size does not divide up evenly into the dimensions of the Tensor Slice. For example, to compute a 7×7 int8 matrix-matrix dot product using the Tensor Slice, 15 out of the 64 PEs will be effectively wasted. But with the large matrix sizes commonly required for DL applications, this performance loss is not a significant issue in real-world applications.

5.3 Comparison with Intel AI Tensor Block

Intel Stratix NX FPGAs contain a new block called the AI Tensor Block [18, 20]. The AI Tensor Block is a replacement of the DSP Slice, in that it exactly matches the I/Os and the area of a DSP Slice. Each Tensor Block contains three dot product units, each of which has 10 multipliers and 10 accumulators. A Tensor Block has 7.5× more int8 compute compared to an Intel Agilex DSP Slice. Multiple blocks can be cascaded to support larger matrices.

AI Tensor Blocks are similar to Tensor Slices in that both these blocks are integrated into the programmable logic. However, there are many differences. We perform a qualitative and quantitative comparison of the two types of blocks below. We use this notation: A matrix-vector multiplication (MVM) involves multiplying M×K matrix with a K×1 vector and a matrix-matrix multiplication (MMM) involves multiplying a M×K matrix A with a K×N matrix B.

(1) AI Tensor Blocks are smaller (1× an actual Stratix 10 DSP) compared to Tensor Slices (4.3× our Agilex-like DSP). Note that our Tensor Slice's implementation is full-featured, but our DSP Slice does not have all the features that are present in Intel's DSP slices (as mentioned in Section 4.2). As an approximation, 4.3x can be used as the ratio of the area of a Tensor Slice and an AI Tensor Block. An AI Tensor Block has 30 int8 MAC units, but a Tensor Slice

has 64 int8 MACs. AI Tensor Blocks have lower compute throughput compared to Tensor Slices, but higher throughput per unit area.

(2) Information about AI Tensor Block operating frequency is not available in the Intel Stratix 10 datasheet. In [20], the frequency used for calculating peak throughput is 600MHz, but it is not clear if this is the maximum frequency of operation of the AI Tensor Block. We optimistically assume it to be the same as the Stratix 10 DSP Slice, which is 1000 MHz for fixed-point and 750 MHz for floating-point (in 14nm technology node). Scaling to 22nm (which is the technology node, we've used for our evaluation) using equations from [27], this comes out to be 442 MHz for fixed-point and 331 MHz for floating-point. This is faster than the Tensor Slice.

(3) Tensor Slices and AI Tensor Blocks support different set of precisions. The base precisions supported on AI Tensor Block are int8 and int4. There is no native floating point support, but instead, there is shared exponent support for block floating point fp16 and fp12 [30]. On Tensor Slices, we completely support int8, int16, fp16 and bf16 precisions. The only common precision between the two is int8.

(4) Tensor Slice and AI Tensor Block have different bandwidth requirements. That is, the number of bytes read/written per cycle by each slice to perform a computation is different. For matrix-matrix multiplication, Tensor Slices read the inputs only once because of their systolic architecture. So, the inputs can be streamed in, for example, from the previous layer's logic on the FPGA. However, in case of the AI Tensor Block, one of the input matrices is read multiple times, so it has to be stored and fed to the block. For matrix-vector multiplication, Tensor Slices and AI Tensor Block both only read the inputs once. However, in both cases, to get more speedup, more blocks can be used and that typically involves reading the inputs once and fanning them out to the multiple blocks.

(5) Both Tensor Slice and AI Tensor Block suffer significant under-utilization for matrix-vector multiplication as we will see in the quantitative comparison later in this section. Tensor Slice has a utilization of 25% for int8 precision, but for 16-bit precisions (int16, fp16, bf16), its utilization is 50%. AI Tensor Block has a utilization of 33% for any precision for matrix-vector multiplication. For matrix-matrix multiplication, 100% utilization can be achieved on both the blocks. For the Tensor Slice, rounding up or zero padding the problem size to nearest multiple of 8 is needed for int8 precision and to nearest multiple of 4 for 16-bit precisions. This is true for all three dimensions - M, N, and K. For the AI Tensor Block, rounding up or zero padding to nearest multiple of 6 along the N dimension and nearest multiple of 10 in K dimension leads to maximum efficiency.

For quantitative comparison, we consider two workloads - one is matrix-vector multiplication (MVM) and the second is matrix-matrix multiplication (MMM). We consider the problem size to be such that there are no fragmentation effects in either the AI Tensor Block or the Tensor Slice (i.e. the problem dimensions have to be a multiple of 6, 10, and 8). Table 13 shows the properties and metrics for these workloads. We use int8 precision because that's the only common precision between the two blocks.

We see an interplay between cycles consumed, number of blocks used and the bandwidth requirements. For the MVM workload, we see that Tensor Slice takes a smaller number of cycles, but more blocks are required compared to the AI Tensor Block. More blocks means more routing interconnect will be required and the frequency of operation will likely be lower. The total bytes read during the process is the same (both matrix and vector are read only once). For the MMM workload, we show two implementations which differ in how the blocks are cascaded to perform a large MMM. In implementation #1, only a few AI Tensor Blocks are used compared to Tensor Slices, and

Table 13. Comparing Intel AI Tensor Block (AI.T.Block) with Tensor Slice (T.Slice) for 2 Workloads: A Matrix-vector Multiplication (MVM) and a Matrix-matrix Multiplication (MMM)

	MVM		MMM (imp #1)		MMM (imp #2)	
	AI.T.Block	T.Slice	AI.T.Block	T.Slice	AI.T.Block	T.Slice
M	480	480	480	480	480	480
K	480	480	480	480	480	480
N	1	1	480	480	480	480
Number of blocks used	49	60	49	3600	3920	1800
Approximate area	49	258	49	15480	3920	7740
Clock cycles	674	274	76996	1920	1156	3840
Utilization	33%	25%	100%	100%	100%	100%
Peak i/p BW (elements in 1 cycle)	480	482	490	960	1280	720
Peak o/p BW (elements in 1 cycle)	10	480	30	480	2400	240
Total bytes read	230880	230880	37094400	460800	18892800	691200
Fanout for matrix B (or vector)	1	1	1	1	1	1
Fanout for matrix A	1	2	1	1	80	1

There are two implementations for MMM: imp #1 and imp #2. They differ in how multiple blocks are cascaded. Note that this is only for int8 precision. The area is in terms of number of DSPs and only includes the area of the blocks, not the routing wires.

so the AI Tensor Blocks take a large number of cycles, while also reading a much larger number of bytes (matrix A has to be read multiple times). On the other hand, Tensor Slices take a very small number of cycles, but the number of blocks required is very high. In implementation #2, for the AI Tensor Block case, we use more blocks leading to a significant reduction in cycles, but a much higher fanout requirement for matrix A and also higher input (i/p) and output (o/p) bandwidth is required. Higher fanout generally means a lower frequency of operation and higher bandwidth requirement implies more routing interconnect usage. More blocks also means more routing wires required to connect the blocks, which can nullify the area advantage the AI Tensor Block has over Tensor Slice in this case. In implementation #2 for the Tensor Slice case, we use half the number of blocks compared to implementation #1, which doubles the number of cycles required. Both Tensor Slice based implementations read much smaller number of bytes compared to the AI Tensor Block based implementations because of the systolic architecture of the slice. This implies reduced dynamic power consumption. Tensor Slices consume more area in both implementations.

Let us consider two smaller workloads as well. First one is an MMM of size M = 12, K = 20, N = 12. This workload has no fragmentation when the AI Tensor Block is used. Three AI Tensor Blocks are required, the number of cycles taken is 60, and the total elements read are 1,200. When computed using Tensor Slices, four blocks are required, 80 cycles are consumed, and 768 elements are read. The second workload is an MMM of size M = 16, K = 16, N = 16. This workload has no fragmentation when the Tensor Block is used. Four Tensor Slices are required, the number of cycles taken is 64, and the total elements read are 512. When computed using AI Tensor Block, three blocks are required, 108 cycles are consumed, and 2,280 elements are read.

Overall, there is no clear winner and both blocks have their positives and negatives. There are many ways in which both the blocks can be connected to perform MVMs and MMMs and the best solution depends on the requirements of the application. From the point of view of approximate area (i.e. only block area, not the area of routing wires required to connect the blocks), AI Tensor Blocks are a better choice. Either block can be chosen based on cycles. If bytes read (and dynamic power) is concern, then Tensor Slices are a better option. Note that these comparisons are valid only for int8 precision.

5.4 Comparison with Xilinx AI Engine

Xilinx Versal ACAPs (Adaptive Compute Acceleration Platform) add an array of AI engine tiles [33, 34] to the FPGA chip. Each AI engine tile contains an AI engine and a data memory. The AI engine contains a SIMD (Single Instruction Multiple Data) VLIW (Very Long Instruction Word) vector processor. The data memory is 32 Kilobytes. An AI engine can access the data memory of its near neighbors as well. The AI engine array can communicate with the programmable logic via AXI stream and memory mapped interfaces over a NoC (Network-On-Chip). We perform an analytical comparison of the Tensor Slice and the AI engine in this section:

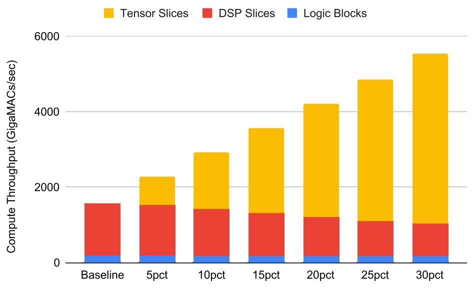
(1) Integrating AI engines on an FPGA is similar to adding CPUs on same die, whereas integrating Tensor Slices on an FPGA is similar to adding DSP slices in the FPGA's programmable logic. AI engines are not tightly coupled with the FPGA programmable logic. They follow a different computation paradigm. This approach is a software centric approach to acceleration. It does make programming the FPGA easier, but it brings with it the overheads of instruction pipelines like addressing, decode, etc., as well as additional overheads in communicating with compute/control units designed in the programmable logic. Tensor Slice, on the other hand, enables tighter integration with any custom logic implemented in the programmable logic.
(2) The precisions supported by the AI engine are 8-bit fixed-point, 16-bit fixed-point, 32-bit fixed-point and IEEE floating point 32-bit. It does not support smaller floating-point precisions like fp16 and bf16, which are commonly used in DL workloads and are supported by the Tensor Slice.
(3) For 8-bit fixed-point operands (16-bit accumulation), the AI engine's peak throughput is 128 MACs/clock. For 16-bit fixed-point operands (48-bit accumulation), the AI engine's peak throughput is 32 MACs/clock. For both these precisions, the throughput of the AI engine is 2× that of a Tensor Slice.
(4) The AI engine tiles work at 1 GHz at 7nm technology node. Scaling to 22nm (which is the technology node, we've used for our evaluation) using equations from [27], this comes out to be around 300 MHz, which is lower than the frequency of the Tensor Slice.
(5) The area of the AI engine is not publically available. However, we know that in addition to raw MACs, it has a lot of additional control logic like load and store units, instruction fetch and decode units, debug and trace units, etc. The Tensor Slice, on the other hand, has a high ratio of compute to control logic. With the additional logic in the AI engine and given that it has 128 int8 MACs, it is likely much larger than 2× the area of a Tensor Slice. Therefore, the Tensor Slice has a higher throughput per unit area compared to an AI engine.

6 RESULTS

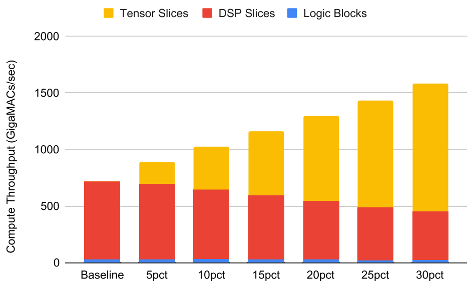
6.1 Peak Throughput

One of the most important benefit of adding Tensor Slices to FPGAs is to increase the compute density. That is, an FPGA with Tensor Slice has more compute throughput per unit area than a baseline FPGA. In this section, we evaluate the peak throughput of the baseline and the proposed FPGAs.

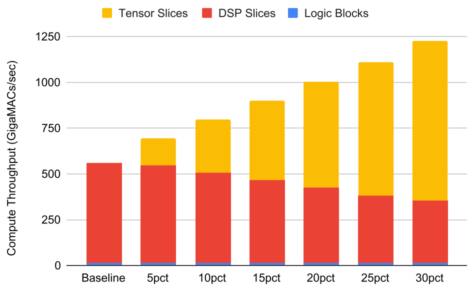
To evaluate the peak throughput, we consider the MAC (multiply-accumulate) operation, which is the most common operation in DL applications. For LBs, one MAC is implemented on the FPGA to determine the operating frequency and number of LBs consumed. Using this and the total number of LBs on the FPGA, we calculate the total number of MACs that can be implemented on the FPGA using LBs. For DSPs, we multiply the number of MACs that can be implemented on one DSP Slice (for a given precision) with the number of DSPs on the FPGA to find the total number



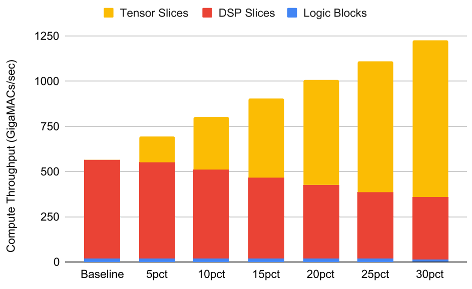
(a) INT8



(b) INT16



(c) FP16



(d) BF16

Fig. 10. Compute throughput of the full FPGA for each precision increases as we increase the percentage of area consumed by Tensor Slices on the FPGA. 5pct, 10pct, etc. denote variants of proposed FPGA architecture. The letters "Prop_" are omitted for brevity.

of MACs. For Tensor Slices, the method to evaluate the total throughput is same as the method for DSP Slices. Then we add up the throughput from all compute resources (LBs, DSPs, and Tensor Slices) to find the total peak MAC throughput of the FPGA. Note that while doing this calculation, we assume that the frequency of operation when the FPGA is filled with MACs using a compute unit is the same as the operating frequency of one MAC for that compute unit. This ignores the frequency degradation as the FPGA is filled, but serves the purpose of evaluating peak throughput.

Figure 10 shows the peak throughput for each precision supported by the Tensor Slice, obtained from each different computing resource in GigaMACs per second. We see significant increase in the throughput by spending some area of the FPGA on Tensor Slices. For example, for the Prop_10pct case, 10% of the area of the FPGA is spent on Tensor Slices, and we see that the peak compute throughput increases by 1.86× for int8 and ~1.42× for int16, fp16, and bf16 precisions. For Prop_30pct variation of the proposed architecture, the throughput increases by 3.5× for int8, 2.2× for in16, 2.18× for fp16 and 2.17× for bf16.

6.2 Resource Usage

Table 14 shows the resource usage obtained from VTR for the various benchmarks when implemented on the baseline and proposed FPGAs. For DL benchmarks, we can see that the usage of LBs and DSPs reduces greatly with the usage of Tensor Slices. The highest reduction in LB usage was in `tpuld.32` with 0.10× usage (i.e. 90% reduction from baseline architecture). The same number of blocks are used by the benchmarks across the variants of the proposed FPGA; so we only show one column for Proposed. For non-DL benchmarks, we do not see any difference in the resource usage between baseline and proposed FPGAs. This is because the designs do not have any matrix operations in them and use the same blocks in both types of FPGAs. A larger percentage of Tensor

Table 14. Resource Usage of Various Benchmarks

	Logic Blocks		DSP Slices		RAM Blocks		Tensor Slices	
Benchmark	Baseline	Proposed	Baseline	Proposed	Baseline	Proposed	Baseline	Proposed
`arm_core`	836	836	0	0	40	40	0	0
`bgm`	2132	2132	11	11	0	0	0	0
`blob_merge`	540	540	0	0	0	0	0	0
`LU32PEEng`	5890	5890	32	32	274	274	0	0
`LU8PEEng`	1728	1728	8	8	73	73	0	0
`mcml`	6792	6792	106	106	294	294	0	0
`or1200`	202	202	4	4	4	4	0	0
`stereovision0`	582	582	0	0	0	0	0	0
`stereovision1`	490	490	40	40	0	0	0	0
`stereovision2`	1958	1958	483	483	0	0	0	0
`attention`	1706	757 (0.44×)	73	9	188	188	0	16
`conv.fp`	630	243 (0.38×)	75	0	56	56	0	7
`conv.int`	913	243 (0.26×)	42	0	56	56	0	7
`eltadd`	287	71 (0.25×)	0	0	24	24	0	2
`eltmul`	630	206 (0.32×)	96	0	48	48	0	6
`fcl.bf`	2017	788 (0.39×)	200	0	0	0	0	25
`fcl.int`	464	97 (0.20×)	112	0	24	24	0	4
`lstm`	6982	1883 (0.27×)	642	2	532	532	0	160
`tpuld.16`	744	134 (0.18×)	148	0	14	14	0	4
`tpuld.32`	2440	258 (0.10×)	552	0	26	26	0	16

Resource usage is the same for all variants of the proposed FPGA architecture, hence there is only one moniker used here: "Proposed".

Slice area can lead to insufficient resources for a large non-DL design, and hence the design may not fit. However, for large non-DL designs, it is better to choose an FPGA from a device family that is oriented towards non-DL applications, instead of using a DL-optimized FPGA that has Tensor Slices.

6.3 Area

The total area consumed by a circuit on an FPGA is the sum of the logic area and the routing area. Logic area is available in the VTR output report, but routing area is not. The routing area is estimated approximately by adding the routing area of all tiles that had at least one operation mapped to it.

For DL benchmarks, the total used area reduces significantly (Figure 11). This follows directly from the usage of Tensor Slices instead of LBs and DSP slices. On average, the area reduces to 0.45×. The best case we see is 0.22× for `fcl.int`. Tensor Tiles harden the circuitry that would otherwise be implemented in soft logic and DSP slices. These results (along with Routed Wirelength results in Section 6.5) provide a first order approximation of the potential power reduction that can be achieved by using Tensor Slices. The area reduction does not differ significantly for different variations of the proposed FPGA architecture, similar to resource usage.

Non-DL benchmarks do not show any change in total area. Logic area is not expected to change because the resource usage does not change. The routing area may change slightly because of the presence of Tensor Tiles. However, our routing area model is approximate as it only considers the routing area of the tiles that had at least one operation mapped to, which also stays constant for

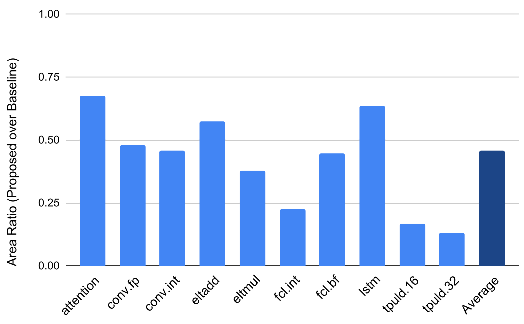


Fig. 11. Comparison of used area for various benchmarks. Significant area reduction is seen in DL benchmarks (average = 55%). Non-DL benchmarks not shown because of no change in their area.

non-DL benchmarks. Therefore, we only show the benefits to total area for DL benchmarks in Figure 11.

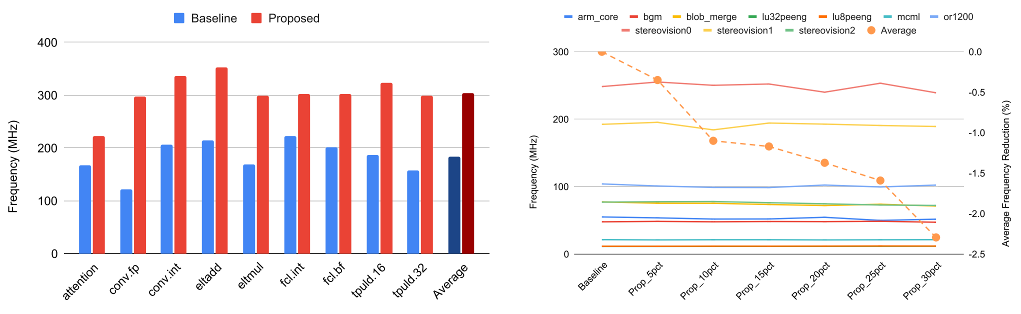
6.4 Frequency

Figure 12(a) shows the improvement in the frequency of operation of DL benchmarks on a proposed FPGA compared to the baseline FPGA. Increase in frequency implies a reduction in execution time at the application level. We see a maximum frequency improvement of 2.43× for the `conv.fp` benchmark and an average improvement of 1.63× across benchmarks. This boost happens because on the baseline FPGA, the critical paths include long paths across LBs and DSP slices, which are inside the hard Tensor Slice on the proposed FPGA. The achieved frequency does not change significantly across different variations of the proposed FPGA architecture (i.e. with different area percentage consumed by Tensor Slices on the FPGA), so we only show results for one variation in the figure.

In non-DL benchmarks, the frequency degrades when Tensor Slices are added, although not significantly. Some degradation is expected because the presence of Tensor Slices can increase the routing wire length required to route a circuit causing an increase in the critical path delay. Figure 12(b) shows the sensitivity of frequency of operation for non-DL benchmarks as the area spent on Tensor Slices increases in the different variations of the proposed FPGA. An average degradation of 2.3% is observed for Prop_30pct variation of the proposed architecture. The maximum degradation observed was 7.9% in the `blob_merge` benchmark. Reduction in frequency implies increase in execution time. If a large non-DL design can not utilize Tensor Slices (e.g. in Individual PE mode), then it may not fit on an FPGA where a large portion of the area is consumed by Tensor Slices. To fit the design on the FPGA, some parallelization may have to be reduced (if possible, based on the nature of the design), leading to the application consuming more time. However, for large non-DL designs, it is better to choose an FPGA from a device family that is oriented towards non-DL applications, instead of using a DL-optimized FPGA that has Tensor Slices.

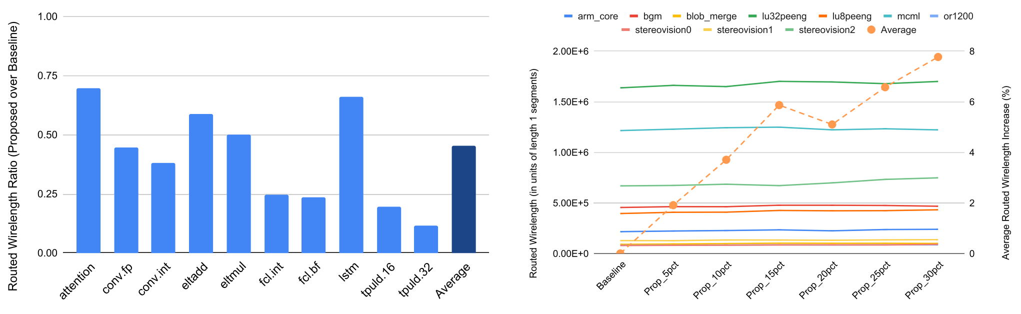
6.5 Routed Wirelength

Figure 13 shows the impact of using Tensor Slices on the total routed wirelength for various benchmarks. For DL benchmarks (Figure 13(a)), we see a significant reduction in routed wirelength. This is because a lot of wiring required to connect soft logic and DSP slices on the baseline FPGA is effectively absorbed inside the hard Tensor Slice. On average, the routed wirelength reduces to 0.45× (55% reduction). In some circuits where the portion of the design that is mapped to Tensor Slices is large, the routed wirelength reduction is very high. E.g. for `tpuld.32` design, the routed wirelength reduces by ~90%. In other designs which have significant portion of the design that is



(a) DL benchmarks **(b)** Non-DL benchmarks

Fig. 12. Comparison of achieved frequency of operation for various benchmarks. DL benchmarks benefit significantly (average = 65%). Non-DL benchmarks show a slight degradation, which increases as the area occupied by Tensor Slice increases.



(a) DL benchmarks **(b)** Non-DL benchmarks

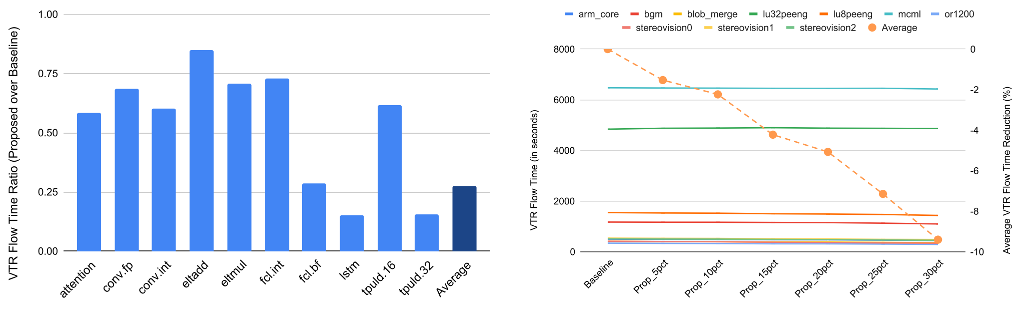
Fig. 13. Comparison of routed wirelength used for various benchmarks. The routed wirelength decreases by an average of 55% for DL benchmarks and slightly increases for non-DL benchmarks.

not mapped to Tensor Slices, the routed wirelength reduction is low. E.g. wirelength reduction of ~35% and ~31 % is seen in `lstm` and `attention` designs, respectively. These results (along with Area results in Section 6.3) provide a first order approximation of the potential power reduction that can be achieved by using Tensor Slices. Different variations of the proposed FPGA show similar reduction in routed wirelength and hence, they are not shown in the figure.

For non-DL benchmarks (Figure 13(b)), the routed wirelength increases slightly. Adding a large block on the FPGA can increase wire length required to route a design that does not use the large block. We see an average increase of 1.9% across non-DL benchmarks for the Prop_5pct FPGA architecture. As we increase the percentage of area consumed by the Tensor Slices in the variations of the proposed FPGA architecture, the routed wirelength increases. That's because longer net lengths are required to make connections between blocks with an increasing number of Tensor Slices in the fabric. An average increase of 7.7% in routed wirelength is seen in the Prop_30pct FPGA architecture.

6.6 VTR Flow Run Time

Tensor Slices are large hard blocks that make the FPGA more coarse-grained. A design using Tensor Slices can go through the FPGA tool chain (synthesis, packing, placement, routing) faster. Having Tensor Slices in the FPGA fabric reduces the total number of blocks on the FPGA (for the same area), and hence reduces exploration space available to the packing, placement and routing algorithms, making the runtime shorter. Also, Tensor Slices are instantiated as hard macros in the RTL. There is



(a) DL benchmarks **(b)** Non-DL benchmarks

Fig. 14. Comparison of VTR flow run time for various benchmarks. The runtime decreases by an average of 73% for DL benchmarks and by up to 9.4% for non-DL benchmarks.

no behavioral code that is analyzed to infer Tensor Slices. This reduces the runtime of the synthesis engine.

Figure 14 shows the VTR CAD flow time for various benchmarks. We see that the time taken to run the VTR CAD flow reduces by an average of 73% for DL benchmarks when using Tensor Slices. Since the time taken does not change significantly across different variants of the proposed FPGA architecture, we do not show the results for all variants in the figure. For non-DL benchmarks, we see a trend in the time taken by the CAD flow. As the percentage of area consumed by Tensor Slices on the FPGA increases, the time taken by the flow reduces. A reduction of up to 9.4% is seen for the Prop_30pct architecture.

6.7 Routing Channel Width

In all the results discussed so far, a fixed routing channel width of 300 is used for all the FPGA architectures. In this section, we conduct experiments in which we vary the routing channel width to study the impact of adding Tensor Slices on channel width. VTR provides a mode where it finds the minimum channel width required to route a circuit on a given FPGA architecture. Minimum channel width requirement is influenced by the pin density of the building blocks in an FPGA architecture. Blocks with small area and large number of I/Os can cause routing congestion and hence lead to higher minimum channel width requirement. A large block with a large perimeter (or area), like the Tensor Slice, does not negatively impact the routability, even with many I/Os. If a block disrupts the routing fabric by not allowing wiring to cross it, then that can cause routability issues as well. We define the Tensor Slice tiles to have full switch boxes outside and straight switch boxes inside the block [22], alleviating the routability impact. Note that this requires careful physical layout of the Tensor Slice block. Adding a local crossbar inside the Tensor Slice blocks helps with improving the routability of the proposed FPGA as well.

Figure 15 shows the minimum routing channel width required for various FPGA architectures, averaged across the benchmarks. We see that DL benchmarks have a lower channel width required compared to non-DL benchmarks, even for the baseline FPGA. As the area consumed by Tensor Slices increases (in variations of the proposed FPGA architecture), the channel width requirement reduces. This is because of the more spread out placement of the blocks used for the circuits because of the presence of more and more Tensor Slices (which leads to increase in routed wirelength, as we saw in Section 6.5).

6.8 FPGA Grid Area

In all the results discussed so far, fixed FPGA grid areas (number of columns × number of rows) were used. The grid dimensions were calculated such that the FPGA is capable to fit the total

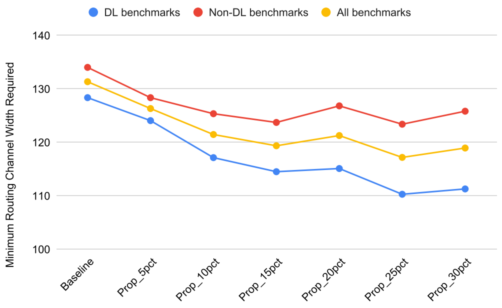


Fig. 15. Minimum routing channel width required for the baseline FPGA architecture and the different variations of the proposed FPGA architecture, averaged across different categories of benchmarks.

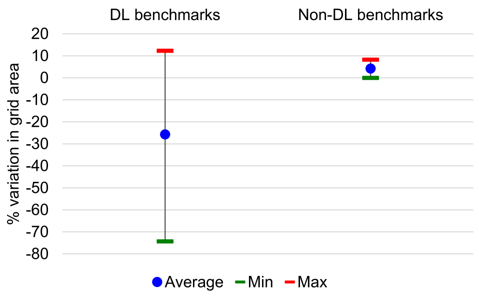


Fig. 16. Variation in the FPGA grid area observed when comparing Proposed(Auto) FPGA architecture relative to the Baseline(Auto) FPGA architecture, across DL and non-DL benchmarks.

number of blocks of each type mentioned in Tables 8 and 9. In this section, we conduct experiments in which we vary the grid dimensions of the FPGA to observe how the minimum FPGA size required for a benchmark changes with the addition of Tensor Slice columns in the FPGA architecture. VTR provides a mode called `auto_layout`, where it expands the grid dimensions (by repeating a specified set of columns and rows) to find the minimum FPGA size required to successfully implement a circuit on the given FPGA architecture. With the introduction of Tensor Slices (i.e. replacing some columns with columns containing Tensor Slices in the FPGA), the total number of columns required to implement a circuit that does not use Tensor Slices can increase. In other words, to fit the same design, a larger FPGA may be required. This can show the impact of adding Tensor Slices especially on non-DL designs.

For this study, we define FPGA architectures in which a set of columns is specified and this set is repeated over and over by VTR to satisfy the resource requirement of a design. For the baseline architecture, this set includes 11 Logic Block columns, four DSP Slice columns, and three RAM Block columns. For the proposed architecture, this set includes 11 Logic Block columns, four DSP Slice columns, three RAM Block columns and one Tensor Slice column. These architectures are referred to as Baseline(Auto) and Proposed(Auto), respectively.

Figure 16 shows the results from this study. We observe that for DL benchmarks, the average grid area required reduces by 26% when the Proposed(Auto) architecture is used, compared to when the Baseline(Auto) architecture is used. This is because DL benchmarks use Tensor Slices and a smaller number of columns are now required to fit the same design. The minimum reduction in FPGA grid area required is seen to be 74.2%. We see an interesting behavior though. For DL benchmarks, we observe that the FPGA grid area required increases in some cases, the maximum increase being 12.3%. This happens because of the arrangement of Tensor Slice columns in the Proposed(Auto) architecture. As the FPGA grid size is increased by VTR, to get an additional Tensor Slice column, the grid size has to be increased by 19 columns, even if 18 out of 19 columns may not be utilized because they contain other resources. The Tensor Slice columns are equally spread out in the architecture. This illustrates the importance of having Tensor Slice columns close to each other like we have in our Prop_Xpct architectures, instead of being spread out on the entire FPGA. For non-DL benchmarks, however, the average grid area required increases by 4.3% when the Proposed(Auto) is used. A larger FPGA is now required to fit all the designs.

6.9 Non-DL Benchmarks when Sufficient DSP Slices are Not Available

Adding Tensor Slices on an FPGA takes away area from other resources like Logic Blocks, DSP Slices, and RAM Blocks. This implies that for large non-DL designs requiring many DSP Slices,

Table 15. Tensor Slices Used by a Non-DL Design in Individual PE Mode When Sufficient DSP Slices are Not Available

Block	Baseline	Prop_30pct
Logic Blocks	3107	2713
RAM Blocks	0	0
DSP Slices	599	298
Tensor Slices	0	108

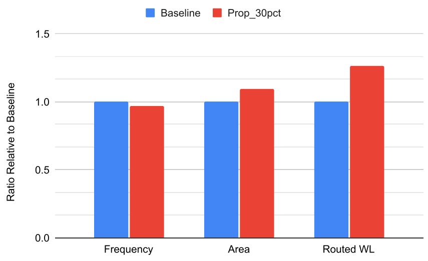


Fig. 17. Comparing various metrics for a non-DL design when implemented on a baseline FPGA using DSP Slices vs. proposed FPGA using Individual PE mode of Tensor Slices for some compute.

the number of DSP Slices in a proposed FPGA may be less than the number of DSP Slices required by the design. In such cases, Tensor Slice's Individual PE mode can be used. Note that the Individual PE mode of the Tensor Slice is not as performant as a DSP Slice, because of the lower frequency of operation of the Tensor Slice compared to the DSP Slice, and it supports smaller precisions only (int8, int16, fp16, and bf16) compared to larger precisions supported by DSP Slice (e.g. 27×27 and fp32). In this section, we evaluate the impact of using Tensor Slice's Individual PE mode.

Since none of the benchmarks in the VTR benchmark suite is very DSP-intensive, we create a synthetic benchmark by instantiating multiple stereovision designs (one stereovision2 instance and three stereovision1 instances). This design requires 599 DSPs on the baseline FPGA (Table 15). We consider the worst case by using the Prop_30pct FPGA for this experiment. In the Prop_30pct FPGA, there are only 500 DSP Slices. So, when this design is implemented on the Prop_30pct architecture, some operations get mapped to the Tensor Slice (in Individual PE mode). Table 15 shows that 298 DSP Slices get used and 108 Tensor Slices get used.

The results of this study are shown in Figure 17. A frequency degradation of 3% is seen, whereas the area increases by 9% and the routed wirelength increases by 25%. This shows the usefulness of the Individual PE mode of the Tensor Slice. This design would not have otherwise fit on this FPGA. But because of Individual PE mode, this design can be implemented on the proposed FPGA (even the one that has 30% area spent on Tensor Slices) with a minor performance degradation. The increase in area and routing wirelength actually indicates an improved utilization of the FPGA in this case, because these resources would have been lying idle otherwise.

6.10 DL Benchmarks when Sufficient Tensor Slices are Not Available

In our experiments, we evaluate different variations of the proposed FPGA in which the area of the FPGA spent on Tensor Slices increases from 5% to 30% in increments of 5%. A question arises: What happens when the number of Tensor Slices on the FPGA is less than the Tensor Slices required by a design? The answer is that computation has to be mapped to DSP Slices and Logic Blocks instead. In this section, we study the impact on various metrics in such a scenario.

Table 16. Resource Usage of the lstm Benchmark for the Baseline Architecture and Different Variations of the Proposed Architecture

Block	Baseline	Prop_5pct	Prop_10pct	Prop_15pct	Prop_20pct	Prop_25pct	Prop_30pct
Logic Blocks	6982	6285	5388	4479	3724	2816	1883
RAM Blocks	532	532	532	532	532	532	532
DSP Slices	642	562	450	349	242	130	2
Tensor Slices	0	20	48	76	100	128	160

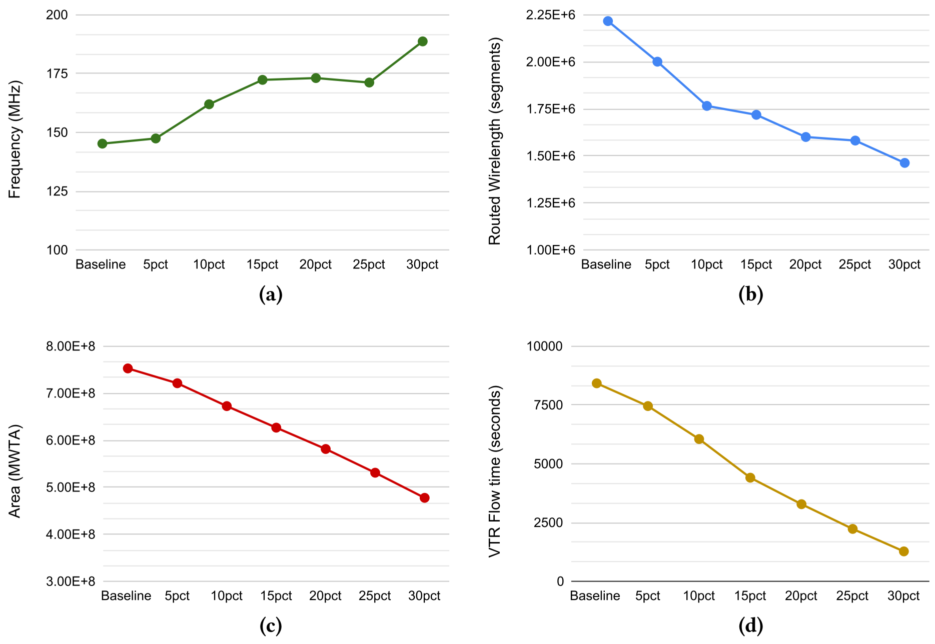


Fig. 18. When a sufficient number of Tensor Slices are not available on an FPGA, some computation of a DL benchmark can be mapped to DSP Slices. As more Tensor Slices become available, more computation can be mapped to them and the performance improves (shown in terms of Frequency, Routed Wirelength, Area and VTR Flow time). 5pct, 10pct, etc. denote variants of proposed FPGA architecture. The letters "Prop_" are omitted for brevity.

We consider the lstm benchmark which requires the largest number of Tensor Slices. We implement this design on the baseline architecture and different variations of the proposed architecture. On the baseline FPGA, all computation is mapped to DSP Slices and Logic Blocks. More and more computation is mapped onto Tensor Slices depending on the number of available Tensor Slices in the variations of the proposed architecture. On the Prop_30pct FPGA, all computation that could be mapped to Tensor Slices is mapped to Tensor Slices. This is shown in Table 16.

Figure 18 shows the results from this experiment. When more computation is mapped to DSP Slices, (1) the frequency is lower, (2) more area of the FPGA is consumed to implement the circuit, and (3) more routed wirelength is used. We learn that when enough Tensor Slices are not available, computation can be mapped to DSP Slices and Logic Blocks at the cost of reduced performance.

7 CONCLUSION

This paper proposes adding Tensor Slices to FPGAs to supercharge their performance for DL workloads. The Tensor Slice efficiently performs common operations used in today's neural networks like matrix-matrix multiplication, matrix-vector multiplication and element-wise matrix addition, subtraction and multiplication. Converting about 10% of the area of our Intel Agilex-like baseline FPGA to Tensor Slices increases the peak compute throughput (GigaMACs/sec) by 1.86× for 8-bit fixed point precision and ~1.42× for 16-bit fixed-point, IEEE Half-Precision Floating Point (fp16), and Brain Floating Point (bf16) precisions. We observe that adding Tensor Slices on the FPGA significantly benefits DL benchmarks in terms of metrics like frequency, area, routing wirelength, etc. On an FPGA architecture with Tensor Slices, we observe 1.63× improvement in frequency and a 55% reduction in area and routing wirelength averaged across several DL benchmarks, compared to an Intel Agilex-like baseline FPGA. We also study the impact of adding Tensor Slices to an FPGA on non-DL applications. We observe a reduction of 2.3% in frequency and an increase of 7.7% in routing wirelength on the FPGA with the most amount of area (30%) spent on Tensor Slices, compared to the baseline FPGA, averaged across several non-DL benchmarks. Product variants of the FPGA with different number of Tensor Slices could be created targeting different application domains. With the abundance of DL applications, adding Tensor Slices to FPGAs is an attractive proposition that will make FPGAs even more appealing for DL acceleration.

REFERENCES

[1] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. 2018. DLA: Compiler and FPGA overlay for neural network inference acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 411–4117. https://doi.org/10.1109/FPL.2018.00077

[2] Achronix. 2019. *Achronix Machine Learning Processor*. https://www.achronix.com/machine-learning-processor.

[3] Achronix. 2021. *Speedster7t FPGAs*. https://www.achronix.com/product/speedster7t/.

[4] Aman Arora, Andrew Boutros, Daniel Rauch, Aishwarya Rajen, Aatman Borda, Seyed Alireza Damghani, Samidh Mehta, Sangram Kate, Pragnesh Patel, Kenneth B. Kent, Vaughn Betz, and Lizy K. John. 2021. Koios: A deep learning benchmark suite for FPGA architecture and CAD research. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. https://doi.org/10.1109/FPL53798.2021.00068

[5] 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

[6] Aman Arora, Zhigang Wei, and Lizy K. John. 2020. Hamamu: Specializing FPGAs for ML applications by adding hard matrix multiplier blocks. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 53–60.

[7] A. Boutros, S. Yazdanshenas, and V. Betz. 2018. Embracing diversity: Enhanced DSP blocks for low-precision deep learning on FPGAs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 35–42.

[8] Fredrik Brosser, Hui Yan Cheah, and Suhaib A. Fahmy. 2013. Iterative floating point computation using FPGA DSP blocks. In *2013 23rd International Conference on Field Programmable Logic and Applications*. 1–6. https://doi.org/10.1109/FPL.2013.6645531

[9] Mohamed Eldafrawy, Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. 2020. FPGA logic block architectures for efficient deep learning inference. *ACM Trans. Reconfigurable Technol. Syst.* 13, 3, Article 12 (June 2020), 34 pages. https://doi.org/10.1145/3393668

[10] Flex-Logix. 2019. *Flex-Logix EFLX eFPGA*. https://flex-logix.com/wp-content/uploads/2019/09/2019-09-EFLX-4-page-Overview-TGF.pdf.

[11] Flex-Logix. 2019. *Flex-Logix nnMAX Inference Acceleration Architecture*. https://flex-logix.com/wp-content/uploads/2019/09/2019-09-nnMAX-4-page-Overview.pdf.

[12] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA'18)*. IEEE Press, Piscataway, NJ, USA, 1–14. https://doi.org/10.1109/ISCA.2018.00012

[13] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx adaptive compute acceleration platform: Versal^{TM} architecture. 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, 84–93. https://doi.org/10.1145/3289602.3293906

[14] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (11 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735 https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf

[15] Intel. 2018. *BFLOAT16 – Hardware Numerics Definition*. https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf.

[16] Intel. 2019. *Intel Agilex FPGAs and SOCs*. https://www.intel.com/content/www/us/en/products/programmable/fpga/agilex.html.

[17] 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.

[18] Intel. 2020. *Intel Stratix 10 NX FPGA Technology Brief*. https://www.intel.com/content/www/us/en/products/programmable/stratix-10-nx-technology-brief.html.

[19] Norman P. Jouppi et al. 2017. In-datacenter performance analysis of a tensor processing unit. *CoRR* abs/1704.04760 (2017). arXiv:1704.04760 http://arxiv.org/abs/1704.04760.

[20] 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

[21] Kevin E. Murray et al. 2015. Timing-driven titan: Enabling large benchmarks and exploring the gap between academic and commercial CAD. *ACM Transactions on Reconfigurable Technology Systems (TRETS)* 8, 2 (2015).

[22] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jai 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.* (2020).

[23] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, R. Krishnamurthy, S. Gribok, B. Pasca, M. Langhammer, D. Marr, and A. Dasu. 2019. Why compete when you can work together: FPGA-ASIC integration for persistent RNNs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 199–207.

[24] Eriko Nurvitadhi, Sergey Shumarayev, Aravind Dasu, Jeff Cook, Asit Mishra, Debbie Marr, Kevin Nealis, Philip Colangelo, Andrew Ling, Davor Capalija, and Utku Aydonat. 2018. In-package domain-specific ASICs for Intel® Stratix® 10 FPGAs: A case study of accelerating deep learning using TensorTile ASIC. *FPGA'18: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 287–287. https://doi.org/10.1145/3174243.3174966

[25] NVIDIA. 2017. *NVIDIA TESLA V100 GPU ARCHITECTURE*. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[26] S. Rasoulinezhad, H. Zhou, L. Wang, and P. 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.

[27] 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/.

[28] Arizona State University. 2012. *Predictive Technology Model*. http://ptm.asu.edu/.

[29] 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.

[30] Wikipedia. 2021. *Block Floating Point*. https://en.wikipedia.org/wiki/Block_floating_point.

[31] Wikipedia. 2021. *Rounding*. https://en.wikipedia.org/wiki/Rounding.

[32] Xilinx. 2018. *Accelerating DNNs with Xilinx Alveo Accelerator Cards*. https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf.

[33] Xilinx. 2018. *Xilinx AI Engines and Their Applications*. https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf.

[34] Xilinx. 2021. *Xilinx ACAP AI Engine Architecture Manual*. https://www.xilinx.com/support/documentation/architecture-manuals/am009-versal-ai-engine.pdf.

[35] Sadegh Yazdanshenas and Vaughn Betz. 2019. COFFE2: Automatic modelling and optimization of complex and heterogeneous FPGA architectures. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12, 1 (January 2019), 3:1–3:27.

Received October 2021; revised January 2022; accepted March 2022