Tensor Slices to the Rescue: Supercharging ML Acceleration on FPGAs

Aman Arora
aman.kbm@utexas.edu
University of Texas
Austin, Texas, USA

Samidh Mehta
samidh99@gmail.com
BITS Pilani
Goa, India

Vaughn Betz
vaughn@ece.utoronto.ca
University of Tornoto
Toronto, Ontario, Canada

Lizy K. John
ljohn@ece.utexas.edu
University of Texas
Austin, Texas, USA

ABSTRACT

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 IEEE half-precision (fp16) math in DSP slices, adding hard matrix multiplier blocks, etc. In this paper, we describe replacing a small percentage of the FPGA's programmable logic area with Tensor Slices. These slices are arrays of processing elements at their heart that support multiple tensor operations, multiple dynamically-selectable precisions and can be dynamically fractured into individual adders, multipliers and MACs (multiply-and-accumulate). These tiles have a local crossbar at the inputs that helps with easing the routing pressure caused by a large slice. By spending ~3% of FPGA's area on Tensor Slices, we observe an average frequency increase of 2.45x and average area reduction by 0.41x across several ML benchmarks, including a TPU-like design, compared to an Intel Agilex-like baseline FPGA. We also study the impact of spending area on Tensor slices on non-ML applications. We observe an average reduction of 1% in frequency and an average increase of 1% in routing wirelength compared to the baseline, across the non-ML benchmarks we studied. Adding these ML-specific coarse-grained hard blocks makes the proposed FPGA a much efficient hardware accelerator for ML applications, while still keeping the vast majority of the real estate on the FPGA programmable at fine-grain.

KEYWORDS

FPGA; neural networks; deep learning; machine learning; hardware acceleration; computer architecture; tensor slice

ACM Reference Format:
Aman Arora, Samidh Mehta, Vaughn Betz, and Lizy K. John. 2021. Tensor Slices to the Rescue: Supercharging ML Acceleration on FPGAs. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28-March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3431920.3439282

1 INTRODUCTION

Machine Learning (ML) has become commonplace in today's world. Owing to the enormous computation requirements of ML and 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 rapid 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 of fine-grained programmable logic blocks (LBs), embedded memory structures (RAMs) 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 multiplication and addition, which are common in DL workloads. This helps pack far more compute in the same area footprint and improves the performance of FPGAs for ML/DL applications.

We architect and implement a Tensor Slice and compare the performance of an FPGA architecture with Tensor Slices (in addition to LBs, DSPs and RAMs) with an FPGA architecture similar to state-of-the-art Intel's Agilex FPGAs. We convert ~3% of the FPGA's area into Tensor Slices resulting in adding only a few columns of Tensor Slices to the FPGA. We observe significant performance boost for ML benchmarks. We explore different percentages of the FPGA area spent on Tensor Slices as well.

FPGAs provide flexibility to meet the requirements of broad range of applications. Adding Tensor Slices to an FPGA is focused on accelerating ML 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-ML applications by causing a higher routing wirelength and longer critical paths. To that end, we also evaluate the impact of this proposal on non-ML applications. We also enhance the Tensor Slice so that it can be fractured into smaller math units like adders, 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 a variety of ML and non-ML 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.

The rest of this paper is organized as follows. The next section provides an overview of related work and compares our proposal with existing solutions. In Section 3, we present architecture of the Tensor Slice. We explain the implementation details of the Tensor Slice, the FPGA used for our experiments, and the benchmarks used for evaluation in Section 4. The results from the experiments we conducted are in Section 5.

2 RELATED WORK

In recent years, many FPGA based hardware accelerators have been proposed and deployed to meet the ever-increasing compute and memory demands of ML workloads. Microsoft's Brainwave [7], Intel's DLA [1], Xilinx's xDNN [18] 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 exist on current FPGAs, such as LBs, DSP slices 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 [8] 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 [14]. Xilinx's Versal ACAP family [19] adds AI engines on the same die as the programmable logic. These AI engines contain vector and scalar processors, with tightly integrated memory. Achronix's Speedster7t FPGAs [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 [6] inference IP contains tiles in which 3×3 Convolutions of Stride 1 are accelerated by dedicated Winograd hardware. Native support for int4, fp16 and bfloat16 data types in DSP slices has also been added to recent FPGAs.

Eldafrawy et al. [5] propose LB enhancements and add a shadow multiplier in LBs to increase MAC density in FPGAs improving DL performance. Boutros et al. [4] have explored enhancing DSP slices by efficiently supporting 9-bit and 4-bit multiplications. Rasoulinezhad et al. [15] suggest improvements to the DSP-DSP interconnect and also inclusion of a register file in the DSP slice to improve data reuse. Nurvitadhi et al. [13] study integrating an ASIC chiplet, TensorRAM, with an FPGA in an SIP to enhance on-chip memory bandwidth.

We propose adding building blocks called Tensor Slices to the FPGAs. Intel recently announced adding Tensor Blocks [10] to the new Stratix 10 NX range of FPGAs. These blocks have 30 MACs and 15x more int8 (8-bit fixed point) compute than a Stratix 10 DSP slice (7.5x compared to an Intel Agilex DSP slice). Our Tensor Slice has 32x more int8 compute compared to Stratix 10 (16x compared to Agilex). Our proposal looks similar to Intel Stratix NX but there isn't much public information available about Tensor Blocks at this point in time to do a comparison with our proposal. Arora et al. [3] propose adding hard matrix multipliers to FPGAs. These blocks support only one precision (fp16 or int8) and one operation (matrix multiplication). The authors do not do an assessment of the impact of adding these blocks on non-ML applications. Also, the implementation details do not mention adding a local input crossbar to the block, which is essential for good routability with a large block.

3 ARCHITECTURE

3.1 Overview

A Tensor Slice is to machine 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 multiplication, along with element-wise matrix addition, subtraction and multiplication. The matrix multiplication operation is 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.

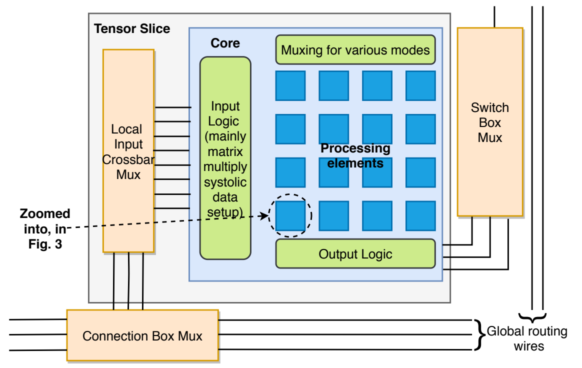

Figure 1: High-level block diagram of the Tensor Slice

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 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 reads input data from RAMs and sends it to the PEs. The output logic shifts out the output data and writes it to a RAM. The muxing logic selects between various modes of operation of the slice.

3.2 Precision

The Tensor Slice currently supports two precisions natively: IEEE half-precision (fp16) and 8-bit fixed-point (int8), although it can extended to support more precisions. These two are the most commonly used precisions in ML inference and training. In the fp16 mode, all multiplications happen in fp16, but accumulations are done in fp32. In the int8 mode, all multiplications happen in int8, but accumulations are done in int16.

3.3 Modes of operation

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 operates on scalar inputs. 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. In other words, the mode bits are not controlled by SRAM cells.

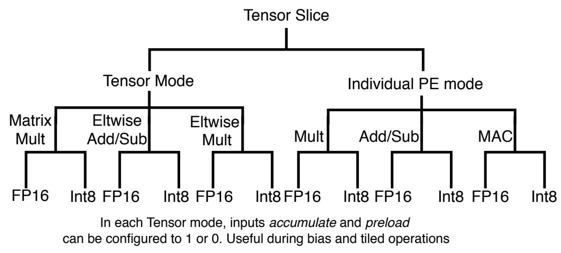

Figure 2: Modes supported by the Tensor Slice

3.3.1 *Tensor Mode.*

In the Tensor Mode (**mode** = 0), the slice can be configured to operate using fp16 precision or int8 precision. In fp16 mode (**dtype** = 1), the Tensor Slice acts on 4x4 matrix operands and generates a 4x4 matrix result. In the int8 mode (**dtype** = 0), the Tensor Slice acts on 8x8 matrix operands and generates an 8x8 matrix result. We observed that by doing this, we can utilize the input-output pins of the Tensor Slice fully in each mode. Also, as we will show in Section 4, the area of the hardware required for a 4x4 fp16 array of processing elements is similar to the area of the hardware required for an 8x8 int8 array, with ample opportunities in sharing the hardware.

There are three sub-modes of the Tensor mode (controlled using **op**): Matrix Multiplication (Dot Product), Eltwise Addition / Subtraction and Eltwise Multiplication. The matrix multiplication operation in the Tensor Slice is done systolically. The elements of first input matrix move from left to right and the elements of the second input matrix 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 is shifted out left to right column-wise in a pipelined fashion. For the eltwise operations, the input and output shifting mechanisms remain the same, but the result calculation happens *after* all inputs have reached their respective locations in the 2D PE array. This part-serial characteristic of eltwise mode elongates the computation time of an eltwise operation, however, it would be serialized with DSP Slices as well because of RAM access. When the results are being shifted out, the another tensor operation can be started on the Tensor Slice.

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.

3.3.2 *Individual PE Mode.*

To reduce the impact of adding Tensor Slices to an FPGA on non-ML applications and to improve utilization, the Tensor Slice supports an Individual PE mode (**mode** = 1). 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 three sub-modes: adder, multiplier or MAC. Furthermore, both fp16 and int8 precisions are available and can be dynamically selected. In the int8 mode, each PE can be configured to be used as 4 8-bit fixed-point multipliers or 4 16-bit fixed-point adders or 4 MACs. In the fp16 mode, each PE can be configured to be used as 1 16-bit floating-point multiplier or 1 32-bit floating-point adder or 1 MAC. Note that because of the large delay in getting to and from the PEs in the Tensor Slice, using the Individual PE mode will not be performant compared to, for example, a LB based addition or a DSP slice based multiplication. The area overhead of adding this mode is discussed in Section 4.3.

The inputs and outputs of an exposed PE are:

- direct_in_a[15:0]
- direct_in_b[15:0]
- direct_mode[1:0] (Multiply, add or mac mode)
- direct_dtype (int8 or fp16 mode)
- direct_out[15:0]
- direct_flags[3:0] (exception flags for floating point mode)

There is a limitation of this mode. The number of inputs and outputs on the slice (also called the IO footprint of the slice) is governed by the Tensor mode (481 inputs, including clock and reset, and 306 outputs). Based on that, 12 PEs out of the 16 PEs have been 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 IO footprint of an already large slice. Increasing the number of IOs can lead to more routing congestion and higher channel width requirement and also a larger area of the Tensor Slice.

3.4 Inputs and Outputs

I/O	Signal	Bits	I/O	Signal	Bits
I	clk	1	I	[a/b/c]_data_in	64*3
I	reset	1	I	valid_mask_[a/b]_rows	8*2
I	mode	1	I	valid_mask_[a/b]_cols	8*2
I	accumulate	1	I	final_op_size	8
I	preload	1	I	address_mat_[a/b/c]	16*3
I	dtype	1	I	address_stride_[a/b/c]	16*3
I	op	1	O	done	1
I	start	1	O	[a/b/c]_data_out	64*3
I	x_loc	8	O	flags	4
I	y_loc	8	O	[a/b/c]_addr	16*3
I	[a/b]_data	64*2	O	c_data_available	1

Table 1: Inputs (I) and outputs (O) of the Tensor Slice

The list of inputs and outputs (IOs) of the Tensor Slice are shown in Table 1. The names of the pins and their description are valid in the Tensor mode (ie. when **mode** input pin is set to 0). Inputs named **mode, accumulate, preload, dtype, op** decide the mode of operation of the Tensor Slice as explained in Section 3.3. As mentioned above, in the Tensor mode, the slice operates on matrix inputs and generates matrix outputs. For this, it reads input matrices from a RAM block. Inputs **address_mat_a** and **address_mat_b** are used to tell the slice the addresses of the first location of each matrix. The slice reads the input matrix A column-wise (it is expected to be stored in column-major order) and reads the input matrix B row-wise (it is expected to be stored in row-major order). The stride inputs (**address_stride_a** and **address_stride_b**) are required to handle the common case where the slice is a part of a larger matrix operation and the address of the first element of the next column/row to read or write is not consecutive. For example, if the slice was being used to multiply a 20x20 matrix with a 20x20 matrix in fp16 mode, then the addresses of consecutive columns and rows will differ by 40, so 40 should be provided to the stride inputs. The slice sends out the address of the location to read on the **a_addr** and **b_addr** outputs. It receives the input data from RAMs on **a_data** and **b_data** inputs. To handle non-square input matrices, the slice provides for providing validity masks for the inputs. This is done using **valid_mask_a_rows**, **valid_mask_a_cols**, **valid_mask_b_rows** and **valid_mask_b_cols** inputs. For example, when multiplying a 6x4 matrix with a 4x7 matrix in int8 mode, the values of these inputs can be 8'b0011_1111, 8'b0000_1111, 8'b0000_1111 and 8'b0111_1111 respectively.

The Tensor Slice performs a tensor operation over multiple clock cycles, depending on the operation. The input **start** is asserted to start the operation and when the operation is done, the slice asserts the **done** signal. The input **address_mat_c** tells the starting address of the output matrix that the slice should write to and the stride in addresses across consecutive columns is specified using **address_stride_c**. While writing, the output data is sent out on **c_data_out** and the address is sent out on **c_addr**. The output **c_data_available** is asserted during this process.

The rest of the inputs and outputs are related to chaining and are explained in Section 3.6.

When the **mode** input pin is set to 1, the Tensor Slice changes to Individual PE mode. In this mode, the usage/meaning of the various pins changes. Each exposed PE's inputs and outputs are mapped onto the top-level inputs and outputs of the slice. 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_mat_mul_size}
- **direct_in_b[15:0]** is mapped to a_data[31:16]
- **direct_mode[1:0]** is mapped to address_mat_c[3:2]
- **direct_dtype** is mapped to address_mat_b[6]
- **direct_out[15:0]** is mapped to c_data_out[31:16]
- **direct_flags[3:0]** is mapped to c_addr[3:0]

3.5 Processing Element

Figure 3 shows the diagram of one processing element (PE) in fp16 mode and int8 mode. There are 16 PEs in the slice and in fp16 mode, the slice needs 16 PEs 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 fp16 mode. For example, logical PE00 in fp16 mode is physical PE00 of the slice, logical PE01 in fp16 mode is physical PE01 of the slice, and so on upto PE33. However, in int8 mode, the slice processes 64 matrix elements, so it needs 64 logical int8 PEs. Because of hardware sharing, each physical PE in the slice acts as 4 logical int8 PEs in int8 mode. So, physical PE00 in the slice maps to logical int8 PE00, PE01, PE02, PE03. Physical PE01 in the slice maps to logical int8 PE04, PE05, PE06, PE07. And so on.

Each PE consists of some 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. The MAC primarily consists of a multiplier and an adder. Hardware is shared between the integer and floating point modes in the multiplier and adder circuits. There is circuitry for 4 int8 multiplications in the multiplier. In fp16 mode, these multipliers are reused (along with additional logic) to perform 1 fp16 multiplication. Similarly, in the adder, there is circuitry for 4 int16 additions, which is reused (along with additional logic) to perform a 1 fp32 addition in the floating point mode.

3.6 Chaining

Multiple Tensor Slices can be chained to perform operations on larger matrices. This is especially important for matrix multiplication operation which is done systolically. Figure 5 shows a logical view of 4 Tensor Slices chained in x and y directions to perform a larger matrix multiplication operation (e.g. a 8x8 matrix multiplied with an 8x8 matrix using 4 slices in fp16 mode). Slice inputs **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. The signals **c_data_in** and **c_data_out** are used for chaining the outputs. Only the Tensor Slices at the periphery interface with RAMs on the FPGA.

In Figure 5, we show different aspects of the Tensor Slice architecture in each slice. The top-left quadrant shows the movement of matrix A elements (in red) and matrix B elements (in yellow) through the PEs. Top-right shows the capture and shift out of the results (i.e. data for matrix C). In the bottom-left part, we show the systolic setup of data from matrix A (left-to-right) for matrix multiplication and the muxing required for chaining. The bottom-right quadrant is the same but for data from matrix B. Note that the figure shows a logical connectivity of the slices in the x and y directions. Physically, these slices can be anywhere on the FPGA. For example, 4 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. The input **final_op_size** is used to specify the overall size of the matrix operation being performed (in this case, 8). These signals are decoded internally to select which inputs to feed to the PEs (e.g. **a_data_in** or **a_data**) and when to start shifting out the result.

3.7 Speeds and Feeds

Matrix multiplication operation in the tensor mode is the most compute intensive operation done by the Tensor Slice. When using

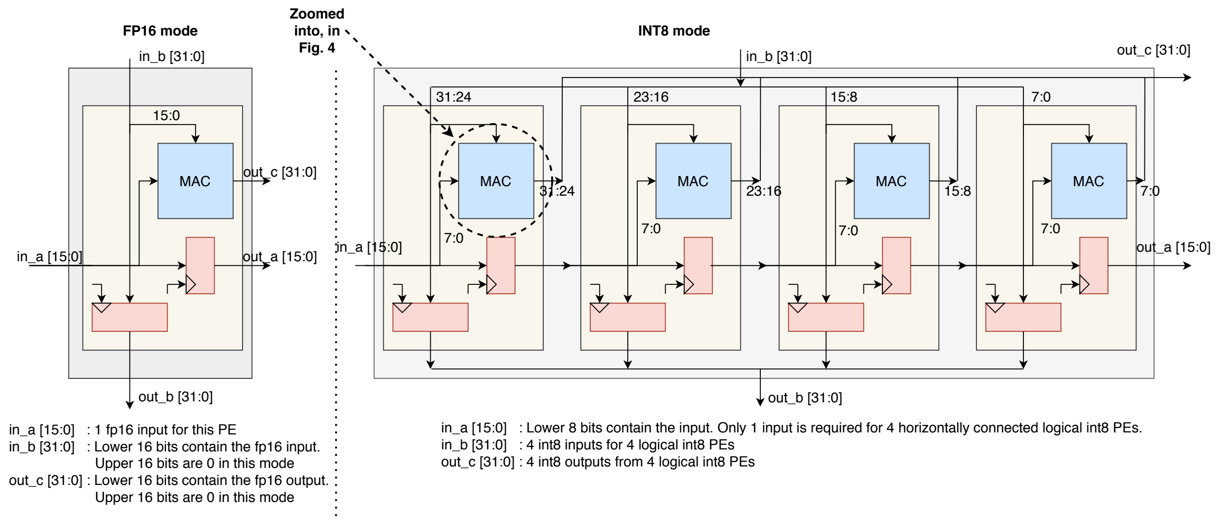
FP16 mode
Zoomed into, in Fig. 4
INT8 mode
in_b [31:0]
15:0
MAC
out_c [31:0]
in_a [15:0]
out_a [15:0]
out_b [31:0]
31:24
23:16
15:8
7:0
in_a [15:0] : 1 fp16 input for this PE
in_b [31:0] : Lower 16 bits contain the fp16 input. Upper 16 bits are 0 in this mode
out_c [31:0] : Lower 16 bits contain the fp16 output. Upper 16 bits are 0 in this mode
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 [31:0] : 4 int8 outputs from 4 logical int8 PEs

Figure 3: A processing element (PE) in FP16 and INT8 modes. There are 16 PEs in the Tensor Slice.

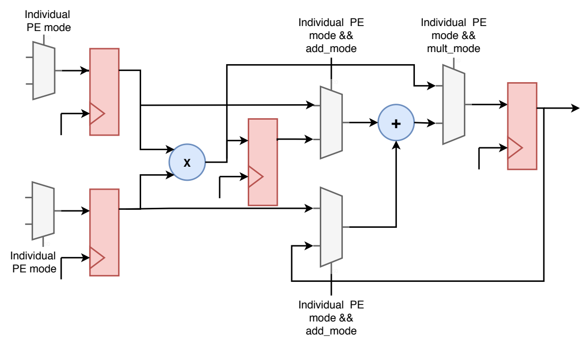
Individual PE mode
Individual PE mode && add_mode
Individual PE mode && mult_mode
x
+
Individual PE mode
Individual PE mode && add_mode

Figure 4: Functional diagram of the MAC block which forms the core of a PE

fp16 precision, the slice performs 16 MAC operations in 1 cycle. So the math throughput of the slice is 16*2=32 fp16 ops/clock. When using int8 precision, the slice's math throughput is 64*2=128 int8 ops/clock. To keep the slice fed with data, it reads 8 fp16 elements every clock cycle in fp16 precision mode 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.

4 EVALUATION

The goal of evaluating the Tensor Slice is to compare the performance of an FPGA with Tensor Slices, LBs, DSP slices and RAMs on it, with an FPGA with only the traditional building blocks (LBs, RAMs and DSP slices). 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.

For our baseline FPGA architecture, we start with a Stratix-10-like architecture used by Eldafrawy et al. [5] and modify it for Agilex.

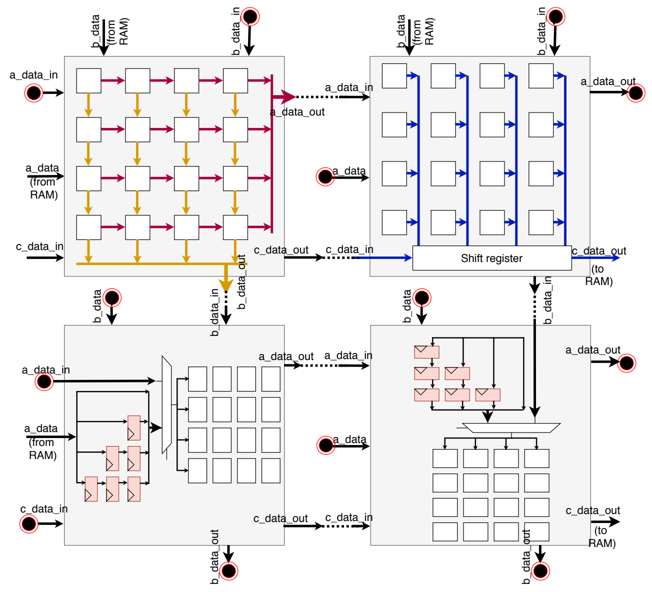

Figure 5: Multiple Tensor Slices can be chained together to perform larger matrix operations. Here, 4 slices are shown to be chained together in x & y direction (logically). Each slice is showing a different piece of detail of the slice internals.

The relevant architectural difference between Stratix-10 and Agilex is the enhanced DSP slice with support for lower precisions like fp16 and fixed-point 9-bit. We implement an Agilex-like DSP slice (details are described later in Section 4.2) and use it in the baseline architecture.

4.1 Tools Used

To evaluate and compare FPGA architectures, we use the Verilog-to-Routing (VTR) tool flow [12]. 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 [20]. 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 [17]. Our standard cell library was the 45nm GPDK library from Cadence. We used scaling factors from Stillmaker et al. [16] 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 [9]. It supports all major modes and all precisions - 9x9, 18x19, 27x27, fp16, fp32. 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 [4]. Our numbers were very similar after scaling for technology nodes. For the Agilex-like DSP slice, the overall delay came out to be 2.97ns in floating point mode and 2.55ns in fixed point mode. Scaling down to 10nm, this is about 740 MHz in floating point mode and 861 MHz in fixed point mode. This is similar to numbers from the Agilex datasheet (750 MHz and 900 MHz respectively).

Table 2 shows the breakdown of the DSP slice area obtained from COFFE. We observe that about 42% of the area of the slice is routing, whereas 58% is the core. The The delay of the 50% sparsely populated local crossbar was 0.333ns. Our DSP slice has 216 non-dedicated inputs and 101 non-dedicated outputs, so it needs access to 4 switchboxes. So, it spans 4 rows in the FPGA grid (1 LB spans 1 row) and the DSP slice column is 1.6x wide (compared to that of a LB). The DSP slice has a IO density of 0.0255 wires per unit area (compared to 0.0158 for the Tensor Slice). Table 3 shows the post-synthesis area of the DSP Slice as we added more precisions to it.

Component	Area (um^2)
Standard-cell core	7089
Local input crossbar	1487
Dedicated output routing	18
Switch box (4)	2752
Connection box	1087
Total	**12433**

Table 2: Breakdown of the DSP Slice area (post P&R)

DSP Slice variant	Area (um^2)	Ratio
Fixed point 18x19 and 27x27 (similar to Intel Arria)	4180	1
Add fp32 support (similar to Stratix 10)	5092	1.22
Add support for fp16 and fixed point 9 bit (similar to Agilex)	5765	1.39

Table 3: Post-synthesis area of the DSP Slice showing overhead of supporting more precisions

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. Because the number of modes supported by the Tensor Slice is quite less compared to the DSP slice (which is designed to be very flexible), the critical path delay of the Tensor Slice was shorter than DSP slice. But because of the much larger number of inputs in the Tensor Slice, the local input crossbar delay was higher (0.723ns). The overall delay came out to be 3.07ns in floating point mode and 2.69ns in fixed point mode. Scaling down to 10nm, the Tensor Slice works at 716 MHz in floating point mode and at 817 MHz in fixed point mode.

Tables 4 and 5 show the breakdown of the Tensor Slice area obtained from COFFE. We observe that about 28% of the area of the slice is routing, whereas 72% is the core. Table 6 shows the post-synthesis area of the Tensor Slice as we added more modes to it. This shows that the creating a Tensor Slice that operates on 4x4 matrices in fp16 mode and 8x8 matrices in int8 mode was justifiable, because there is enough sharing of hardware between the two modes.

Component	Area (um^2)
Standard-cell core	36037
Local input crossbar	6062
Dedicated output routing	0
Switch box (8)	5504
Connection box	2429
Total	**50032**

Table 4: Breakdown of the Tensor Slice area (post P&R)

Component	Area (%age)
Input logic	3.92
Output logic	3.87
2D PE array	92.21
Total	**100**

Component	Area (%age)
Adder	39.2
Multiplier	31.7
Rest	29.1
Total	**100**

Table 5: Area distribution of the various components of the Tensor Slice core (left) and the Processing Element (right)

Our Tensor Slice has 479 non-dedicated inputs and 306 non-dedicated outputs, so it needs access to 8 switchboxes. So, it spans 8 rows in the FPGA grid (1 LB spans 1 row) and the Tensor Slice column is 3.23x wide (compared to that of a LB).

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

Tensor Slice variant	Area (um^2)	Ratio
4x4 fp16 matrix mult only	13338	-
8x8 int8 matrix mult only	16368	1
4x4 fp16 and 8x8 int8 matrix mult	20598	1.26
Add Individual PE modes	24673	1.51
Add Element-wise modes	29062	1.78

Table 6: Post-synthesis area of the Tensor Slice showing overhead of adding more functionality/modes

based on Intel FPGAs and follow from [5]. As a sanity check for our experimental setup, the tile area we obtain from COFFE (1938 um^2) is similar to the one from [5].

When describing the FPGA architecture in VTR, we use a column based layout for all the blocks, similar to modern FPGAs. Our architecture doesn't have sectors though. 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 RAMs). 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, we create an FPGA architecture with the exact same resource mix as Intel AGF 027, in terms of %age of the area and %age 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. Column "Our baseline" in Table 8 shows the total count of each block type, the %age of total area spent on each block type and the %age count of each block type for our baseline FPGA. We then create 3 different FPGA architectures by spending 3% (actually 2.76%), 6% (actually 5.52%) and 9% (actually 8.29%) area of the FPGA on Tensor Slices respectively. These 3 architectures are called "Proposed_3pct", "Proposed_6pct", "Proposed_9pct". The table shows the resource mix of these FPGA architectures as well.

The grid dimensions of all the architectures used in our experiments were around 170x80. The number of columns containing Tensor Slices were 2, 4 and 6 in Proposed_3pct, Proposed_6pct and Proposed_9pct architectures respectively. We placed the Tensor Slice columns in the middle of the FPGA to keep the layout symmetric. Figure 6 shows the layout of the FPGA with 2 Tensor Slice columns.

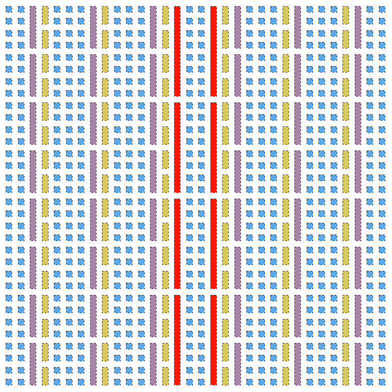

Figure 6: A zoomed-in version of the layout of the Proposed_3pct FPGA architecture obtained from VTR. Blue: LB, Yellow: RAM, Purple: DSP Slice, Peach: Tensor Slice

Parameter	Value	Definition
N	10	Number of BLEs per cluster
W	320	Channel width
L	4	Wire segment length
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 7: Routing and tile architecture parameters of the baseline and proposed FPGA architectures

4.5 Benchmarks

For benchmarking our FPGA architecture, we use a set of ML and non-ML designs. VTR has a benchmark suite [12] that comprises of designs from several applications including finance, computer vision, math and processors. However, this suite currently does not have any ML/DL designs. So, we created a set of ML benchmarks. Table 9 contains a list of all the benchmarks we used. When evaluating an ML benchmark on the proposed FPGAs, Tensor Slices were manually instantiated in Verilog, because the synthesis tool cannot automatically infer them.

4.5.1 TPU-like Design (tpuld). We created a design based on the architecture from Google's TPU v1 [11]. At its heart, it uses a 32x32 matrix multiplication unit, instead of a 256x256 matrix multiplication unit used by the TPU. The design uses int8 precision. The activations are stored in RAM block A, whereas the weights are stored in RAM block B. Control and configuration are done through an APB interface, instead of a PCIe interface on the TPU. The normalization block applies the mean and variance values to the output of the matrix multiplication unit. Pooling unit supports 3 pooling windows - 1x1, 2x2 and 4x4. The activation unit supports two activation functions - rectified linear unit (ReLU) and the hyperbolic tangent (TanH). The activation unit is the last unit before the results are written back to RAM block A, from where they can be read again into the matrix multiplication unit for the next layer.

4.5.2 Fully Connected Layers (fcl8 and fcl16). We created two full-connected layer designs to use as a micro-benchmarks. fcl8 uses int8 precision and has num_features = 15, batch_size = 16 and num_outputs = 14. fcl16 uses fp16 precision and has a 20x20 activation matrix, a 20x20 weight matrix and generates a 20x20 output matrix. The designs contain a state machine to control the operation and a control-and-status register block and a matrix multiplication unit. The inputs and outputs are stored in RAMs on the FPGA.

4.5.3 Element-wise Layers (eltadd and eltmul). We created micro-benchmarks for eltwise layers. eltadd performs eltwise addition of two int8 matrices of size 14x6. eltmul performs eltwise multiplication of two fp16 matrices of size 24x8. The input matrices are read from a RAM block and the result is written to a RAM block.

4.5.4 Convolution Layer (conv). Convolution is a very common operation in ML/DL workloads, especially image recognition and

		Baseline (AG047-like)			Proposed_3pct			Proposed_6pct			Proposed_9pct		
Block	Relative area	# blocks	%age area	%age count	# blocks	%age area	%age count	# blocks	%age area	%age count	# blocks	%age area	%age count
Logic Block	1	8480	45.19	80.92	8240	43.97	80.78	8000	42.74	80.65	7920	42.39	81.15
DSP Slice	6.42	800	27.37	7.63	780	26.72	7.65	760	26.07	7.66	740	25.43	7.58
RAM	4.29	1200	27.44	11.45	1160	26.55	11.37	1120	25.67	11.29	1040	23.88	10.66
Tensor Slice	25.82	0	0	0	20	2.76	0.2	40	5.52	0.4	60	8.29	0.61

Table 8: Resource mix in various FPGA architectures. The Proposed_3pct, Proposed_6pct and Proposed_9pct refer to 3 proposed architectures with approximately 3%, 6% and 9% percentage of FPGA area consumed by Tensor Slices.

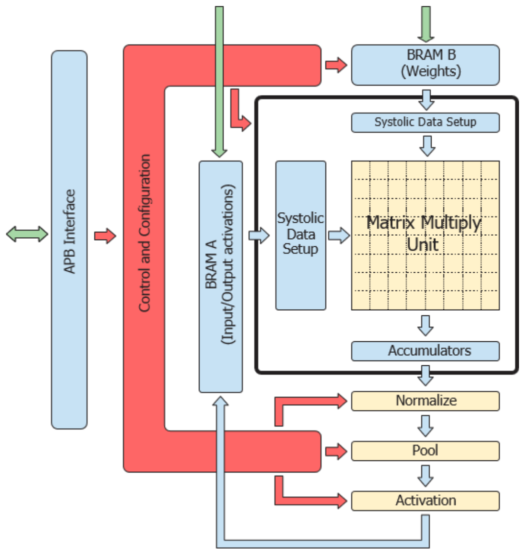

Figure 7: Architecture of the TPU-like design used as an ML benchmark for evaluation

detection networks. We designed a micro-benchmark that convolves a 8x8 image with a 3x3 weight kernel, with padding = 1 and stride = 1. This design uses fp16 precision.

4.5.5 Non-ML benchmarks. For non-ML benchmarks, we use the VTR benchmark suite [12]. We use a variety of benchmarks from the suite such as: mcml, LU32PEEng, stereovision2, LU8PEEng, bgm, stereovision0, and arm_core. These include designs with/without floating point operations, heavy/low DSP usage, and heavy/low/no RAM usage. We used the largest designs from the suite so that the %age utilization of the FPGAs (baseline and proposed) was fairly high to ensure realistic results.

5 RESULTS

5.1 Compute Throughput

Table 10 illustrates a comparison of the throughput of Tensor and DSP slices. The Tensor Slice has 15x int8 and 7.7x fp16 throughput compared to an Agilex DSP slice. An Agilex DSP Slice (baseline) has 4 effective int8 MACs and 2 effective fp16 MACs. On the other hand, a Tensor Slice has 64 int8 MACs and 16 fp16 MACs, albeit at an area cost of ~4x.

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 7x7 int8 matrix-matrix dot product using the Tensor Slice, 15 out of the 64 PEs will be effectively wasted. But with the matrix sizes in use in ML/DL applications, this performance loss is not a significant issue for real-world applications.

5.2 Resource Usage

Table 11 shows the resource usage obtained from VTR for the various benchmarks when implemented on the baseline and proposed FPGAs. For ML 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 fcl16 with 0.09x usage (i.e. 91% reduction from baseline architecture). The same number of blocks are used across the 3 variants of the proposed FPGA; so we only show one column for Proposed. Larger number of Tensor Slices on an FPGA would enable larger ML benchmarks to be implemented without having to use DSP slices.

For non-ML benchmarks, we do not see any difference in the resource usage between baseline and proposed FPGAs. This is explainable because the designs still use the same blocks and they do not have any matrix operations in them. In a case where the design is large enough that it is limited by the number of DSP slices, then the Tensor Slices could be used in Individual PE mode, depending upon the required operations.

5.3 Frequency

Figure 8 shows the improvement in the frequency of operation of the benchmarks when implemented on a proposed FPGA vs. the baseline FPGA. We see that the ML benchmarks achieve a much higher frequency on the FPGA with Tensor Slices. This is expected 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. We see a maximum speedup of 2.82x and an average speedup of 2.44x with the Proposed_3pct architecture. We observe that the achieved frequency is not very sensitive to the area consumed by Tensor Slices. We would have seen a noticeable difference if a benchmark was large enough to be limited by the number of Tensor Slices, and DSP slices would have been required to implement that logic instead. But we do not have such cases in our benchmarks.

In non-ML benchmarks, the frequency remains almost the same, with an average degradation less than 1%. 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. The worst case is 4.2% (in arm_core with

Benchmark	Netlist primitives	Application	Nature of the workload
tpuld	172401	ML/DL	A design similar to Google's TPU v1, with a 32x32 systolic array
fcl8	41596	ML/DL	Fully connected layer. Multiply matrices 14x15 with 15x16 (int8).
fcl16	149272	ML/DL	Fully connected layer. Multiply 20x20 matrices (fp16).
eltadd	22132	ML/DL	A design that adds two 6x14 int8 matrices elementwise
eltmul	48033	ML/DL	A design that multiplies two 24x8 fp16 matrices elementwise
conv	37166	ML/DL	A convolution layer for an 8x8 fp16 input image with 3 channels, padding=1, stride=1, filter size = 3x3 and batch size=2.
mcml	275204	Medical physics	Large design with multiple fixed point multiplications
LU32PEEng	191012	Math	Has floating point operations and extensive RAM usage
stereovision2	51556	Computer vision	Lots of fixed point multiplications and no RAM usage
LU8PEEng	57922	Math	High logic depth, some DSP usage and RAM usage
bgm	111643	Finance	Has floating point operations and no RAM usage
stereovision0	30516	Computer vision	Only LBs, no multiplications in this design, no RAM usage
arm_core	57568	Soft processor	No multiplications in this design, but some RAM usage

Table 9: ML and non-ML benchmarks used for evaluation. Non-ML benchmarks are from the VTR benchmark suite. The netlist primitives are from when the benchmark is implemented on the baseline FPGA with DSP slices.

INT8	DSP Slice	Tensor Slice	Ratio
# Ops	8	128	16
Freq (MHz)	391	371	0.95
Area (um^2)	12433	50032	4.02
Throughput (Gops/sec)	3.13	47.49	15.18
FP16	DSP Slice	Tensor Slice	Ratio
# Ops	4	32	8
Freq (MHz)	336	325	0.97
Area (um^2)	12433	50032	4.02
Throughput (Gops/sec)	1.34	10.4	7.74

Table 10: Comparison of a Tensor Slice with a DSP Slice (all numbers are based on 22nm technology node).

Proposed_9pct). A much larger %age of Tensor Slice area may impact non-ML applications more, but converting a small percentage of the area is sufficient to reap the benefits of Tensor Slices.

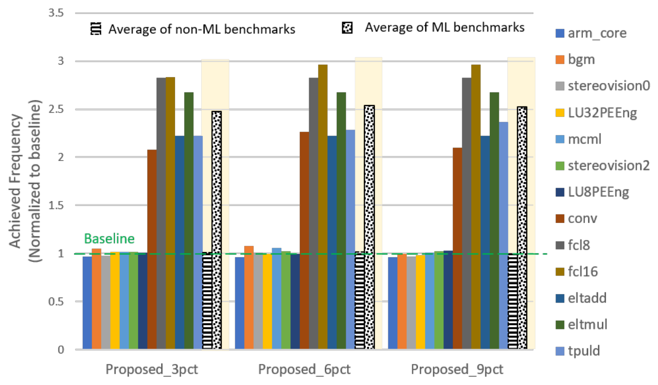

Figure 8: Comparison of achieved frequency of operation for various benchmarks. ML benchmarks benefit significantly, whereas minimal effect in non-ML benchmarks.

5.4 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 atleast one operation mapped to.

For ML benchmarks, the total used area reduces significantly. This follows directly from the usage of Tensor Slices instead of LBs and DSP slices. On average, area reduces to 0.4x. The best case we see is 0.3x for fcl8. Tensor Tiles harden the circuitry that would otherwise be implemented in soft logic. These area results also provide a first order approximation of the potential power reductions that can be achieved when using Tensor Slices instead of DSP slices.

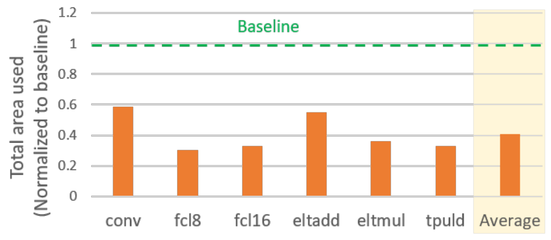

Figure 9: Comparison of used area for various benchmarks. Significant area reduction in ML benchmarks. Non-ML benchmarks not shown because no change in their area.

Non-ML 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 only considers the routing area of the tiles that had at least one operation mapped to, which stays constant for the non-ML benchmarks. Therefore, we only show the benefits to total area for ML benchmarks in Figure 9.

5.5 Interconnect

The impact of adding Tensor Slices on the FPGA's interconnect resources is important because Tensor Slices are large blocks with a large I/O footprint. This can lead to reduction in routability of the FPGA. Adding a local input crossbar to the slice, which we have

	Logic Blocks (LBs)		DSP Slices		RAMs		Tensor Slices	
Benchmark	Baseline	Proposed	Baseline	Proposed	Baseline	Proposed	Baseline	Proposed
arm_core	833	833	0	0	72	72	0	0
bgm	2122	2122	11	11	0	0	0	0
stereovision0	584	584	0	0	0	0	0	0
LU32PEEng	5839	5839	32	32	274	274	0	0
mcml	6788	6788	106	106	270	270	0	0
stereovision2	1931	1931	516	516	0	0	0	0
LU8PEEng	1709	1709	8	8	73	73	0	0
conv	569	226 (0.39x)	42	0	56	56	0	7
fcl8	452	80 (0.17x)	56	0	24	24	0	4
fcl16	1550	145 (0.09x)	200	0	60	60	0	25
eltadd	290	61 (0.21x)	0	0	24	24	0	2
eltmul	627	175 (0.28x)	96	0	48	48	0	6
tpuld	2083	237 (0.11x)	292	0	52	52	0	16

Table 11: Resource usage of various benchmarks. All 3 proposed architectures have the same resource usage, hence there is only one moniker used here: "Proposed"

done as explained in the architecture section, alleviates much of the impact.

Figure 10 shows the impact of adding Tensor Slices on total routing wirelength. For ML benchmarks, we see routing wirelength reducing significantly (~0.25x in the case of fcl16 to 0.6x in case of eltadd). This is expected because a lot of wiring required to connect soft logic on the baseline FPGA is effectively absorbed inside the hard Tensor Slice. As the area spent on Tensor Slices increases, we see a reduction in routing wirelength in benchmarks where Tensor Slices are chained (like fcl16 and tpuld) because instead of connecting Tensor Slices within a column, now Tensor Slices in neighboring columns have to be connected.

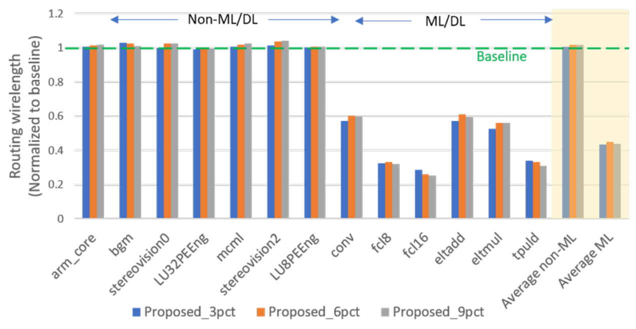

Figure 10: Comparison of routing wirelength used for various benchmarks. The routing wirelength decreases significantly for ML benchmarks and does not degrade for non-ML benchmarks.

For non-ML benchmarks, the routing wirelength increases slightly as expected. 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% across the non-ML benchmarks. As we increase the percentage of area consumed by the Tensor Slices, in most non-ML benchmarks (like mcml, stereovision2, arm_core), the routing wirelength increases or stays the same. There are some aberrations like bgm, which we believe to be noise. A much larger percentage of Tensor Slice area would increase the routing length even more and can even lead to insufficient resources for a non-ML design which can happen for mcml if we convert 20% of the FPGA area to Tensor Slices.

Channel width is another important aspect of routing. With a large block like the Tensor Slice, the required channel width to successfully route a circuit can increase because of the large I/O footprint. We see that in the results reported by VTR. The maximum channel width for successful routing across all benchmarks and FPGA architectures was observed to be 230, which is within the value specified for the FPGA architecture in Table 7 (i.e. 320).

6 CONCLUSION

This paper proposes adding Tensor Slices to FPGAs to supercharge their performance for ML/DL workloads. The Tensor Slice provides 15x int8 compute and 7.7x fp16 compute, compared to an Intel Agilex DSP slice. The Tensor Slice performs common operations used in today's neural networks like matrix multiplication (dot product) and element-wise matrix addition/subtraction and multiplication. We observe that adding a few columns of Tensor Slices on the FPGA significantly benefits ML benchmarks, while non-ML benchmarks show no noticeable degradation. Product variants of the FPGA with different number of Tensor Slices could be created targeting different application domains. With the abundance of ML/DL applications, adding Tensor Slices to FPGAs is an attractive proposition that will make FPGAs even more appealing for ML/DL acceleration, while still keeping FPGAs flexible and performant for non-ML applications.

Although the analysis presented in this paper provides a proof of concept that adding Tensor Slices is beneficial, there are still more explorations to do. We plan to efficiently support more operations in the Tensor Slice like matrix-vector multiplication, to support even smaller precisions like int4 and to provide a more flexible I/O interface.

ACKNOWLEDGMENTS

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.

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. *CoRR* abs/1807.06434 (2018). arXiv:1807.06434 http://arxiv.org/abs/1807.06434

[2] Achronix. 2019. Speedster7t FPGAs. https://www.achronix.com/product/speedster7t/

[3] A. Arora, Z. Wei, and L. 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.

[4] 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)*. 35–357.

[5] 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

[6] 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

[7] 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 (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 1–14. https://doi.org/10.1109/ISCA.2018.00012

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

[9] 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

[10] 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

[11] 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

[12] 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).

[13] 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.

[14] 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

[15] 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.

[16] 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/.

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

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

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

[20] 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.