Hamamu: Specializing FPGAs for ML Applications by Adding Hard Matrix Multiplier Blocks

Aman Arora*, Zhigang Wei†, Lizy K. John‡
Department of Electrical and Computer Engineering
The University of Texas at Austin
*aman.kbm@utexas.edu, †zw5259@utexas.edu, ‡ljohn@ece.utexas.edu

***Abstract*—Designing efficient hardware for accelerating artificial intelligence (AI) and machine learning (ML) applications is a major challenge. Rapidly changing algorithms and neural network architectures make FPGA based designs an attractive solution. But the generic building blocks available in current FPGAs (Logic Blocks (LBs), multipliers, DSP blocks) limit the acceleration that can be achieved. We propose Hamamu, a modification to the current FPGA architecture that makes FPGAs specialized for ML applications. Specifically, we propose adding hard matrix multiplier blocks (matmuls) into the FPGA fabric. These matmuls are implemented using systolic arrays of MACs (Multiply-And-Accumulate) and can be connected using programmable direct interconnect between neighboring matmuls to make larger systolic matrix multipliers. We explore various matmul sizes (2x2x2, 4x4x4, 8x8x8, 16x16x16) and various strategies to place these blocks on the FPGA (Columnar, Surround, Hybrid). We find that providing 4x4x4 hard matrix multiplier blocks in an FPGA speeds up neural networks from MLPerf benchmarks by up to ~3.9x, compared to a Stratix-10 like FPGA with equal number of MACs, same MAC architecture and high DSP:LB ratio. Although the flexibility of the FPGA will reduce for non-ML applications, an FPGA with hard matrix multipliers is a faster, and more area efficient hardware accelerator for ML applications, compared to current FPGAs.**

I. Introduction

Artificial intelligence and machine learning have become ubiquitous in today's world. Algorithms and models for these applications are getting more complex, and data sets are becoming larger and larger. As such, the computation needs are growing exponentially. Accelerating the computation required by AI/ML is a major challenge. Many solutions have been proposed and/or deployed for accelerating deep neural networks in hardware, ranging from ASICs to fully programmable GPUs to configurable FPGA based solutions. ASIC based designs have the best speed and power characteristics (fast and low power), but they lack configurability and adaptability which is crucial in the rapid changing world of AI/ML. GPU and CPU based designs, while highly programmable and adaptable, are not as fast and power-efficient as ASICs. FPGA based designs provide the best of both worlds. They provide massive parallelism, while being flexible and easily configurable, and also being fast and power-efficient.

A question naturally arises: Can we improve the performance of FPGAs for AI/ML? FPGA companies and researchers are exploring and deploying various techniques to make FPGAs better at accelerating AI/ML applications. These range from adding vector processors on the FPGA chip (Xilinx Versal [1]) to providing for integrating custom tensor tiles in the same package (Intel Agilex [2] [3]) to adding support for smaller ML-friendly precisions (like int4, fp16, etc.) in DSP slices.

FPGA devices mainly comprise of fine-grained programmable logic ("soft" LBs), embedded memory structures (RAMs) and fixed-function math units ("hard" DSP slices). Coarser heterogeneous blocks like high speed IO controllers and processors are also seen on many FPGAs. If we look back in time, DSP slices were added to FPGAs when it was realized that designing multipliers and adders using LBs was not efficient and numerous DSP applications required multiplication and addition operations. In the same vein, it is notable that among all the operations executed by state-of-the-art neural networks (like those in MLPerf [4]), about 80-90% of the operations are matrix multiplications (also called GEMM - General Matrix Multiply). Designing a matrix multiplier using LBs and DSP slices leads to a slow and area (hence, power) inefficient implementation [5]. So, in the current era of AI/ML, adding hard matrix multiplier blocks to the FPGAs could have potentially significant benefits. But this brings along several interesting aspects - What should be the size of such blocks? How should they be placed? Should they replace DSP slices? How should they interface with each other and other blocks? In this paper, we explore the answers to these questions.

While the industry and academia have deployed and proposed enhancements to FPGAs for AI/ML, to the best of our knowledge, no one has added hard matrix multiplier blocks to FPGAs. In this paper, we present a modified FPGA architecture, called Hamamu. By converting a small amount of the area of an FPGA to be hard matmul units, we show an increased effectiveness for ML/AI applications while still having sufficient general purpose resources. Although our proposed changes will make the FPGA slightly less flexible, the benefits obtained (presented later in this paper) are large enough to justify making them, especially with the abundance of AI/ML usecases.

Here is the summary of the contributions of this paper:

1) Propose adding hard matrix multiplier blocks to FPGAs and show their benefit for AI/ML applications

2) Derive conclusions regarding the size/dimensions of the hard matrix multiplier blocks
3) Propose adding programmable direct interconnect between hard matrix multipliers and evaluate its benefit
4) Derive conclusions regarding the placement of the hard matrix multipliers on the FPGA

The rest of this paper is organized as follows. The next section provides an overview of related work and mentions how the proposal in this paper is different and what value it adds to the existing solutions. In Section III, we present the proposal Hamamu and its various aspects in detail. The methodology followed to perform the experiments is detailed in Section IV. We discuss the results from these experiments in Section V. We discuss the target of our research, as well as the future work in Section VI, before concluding in Section VII.

II. Related Work

In this decade, many designs and architectures have been proposed and deployed for accelerating AI/ML algorithms, from both the industry and the academia. The focus is on providing more compute resources as well as higher memory bandwidth and efficiency. The Google TPU [6], NVIDIA Volta GV100 GPU [7], DaDianNao [8] are some examples. Many FPGA based solutions exist as well. Microsoft's Brainwave [9] has been deployed in Microsoft's servers to accelerate a multitude of tasks. BrainWave uses Intel's Stratix 10 FPGAs and the design is a soft NPU (Neural Processing Unit) with dense matrix-vector multiplication units at its heart. Xilinx's xDNN FPGA architecture [10] is an overlay processor, containing a systolic array based matrix multiplier, that is mapped onto a generic FPGA. Intel's DLA [11] is also an overlay with a 1-D systolic processing element array at its core which performs dot product operations to implement general matrix math. These FPGA based solutions 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 to make them better for AI/ML applications.

Xilinx recently announced the Versal family of products [1] which adds dedicated AI engine array (SIMD VLIW processors) on the same die as the programmable logic. Intel's latest Agilex FPGAs [2] provide for flexible integration of heterogeneous tiles using Embedded Multi-die Interconnect Bridge (EMIB) in a System-In-Package (SIP). Adding domain-specific accelerator tiles like Tensor Tiles [3] has been explored. Flex-Logix's eFPGAs [12] also support bfloat16 in the MACs in their EFLX tiles, and the MACs can be cascaded without going through the FPGA interconnect. Their nnMAX inference IP [13] contains hard blocks to perform convolutions using the Winograd algorithm. Achronix's Speedster7t FPGAs [14] include Machine Learning Processor (MLP) blocks in the FPGA fabric. These blocks have an array of multipliers, an adder trees and accumulators. They can also be connected to RAMs and other MLP blocks using hard paths. Native support for fp16 and bfloat16 data types in DSP slices has also been added to recent FPGAs. Boutros et al. [15] propose LB enhancements and adds a shadow multiplier in LBs to increase MAC density in FPGAs improving deep learning performance. Boutros et al. [16] and Rasoulinezhad et al. [17] propose DSP slice modifications such as flexible precision and improvements to DSP-DSP interconnect.

In this paper, we propose changing the architecture of FPGAs by adding hard matrix multiplier blocks to the programmable logic part of the FPGA. Nurvitadhi et al. [18] and Lacey et al. [19] make the case for FPGAs as being better than GPUs for machine learning applications. Our proposal further improves the performance of FPGAs, making the argument for using FPGAs for machine learning more compelling.

A matrix multiplier is much larger in size than the usual building blocks on FPGAs (LBs, DSPs, etc). This has interesting complexities, challenges and tradeoffs. FPGAs with coarse-grained units embedded within fine-grained logic blocks are called hybrid FPGAs [20]. Yu et al. [21] discuss architectural tradeoffs involved in adding coarse grained blocks to fine-grained programmable logic on FPGAs. Yu et al. [22] explore routing optimizations for hybrid FPGAs. Shadow clusters are proposed by Jamieson et al. [23], which recover performance loss that happens when large building blocks are unused. Ho et al. [20] discuss modelling and architecture of domain-specific hybrid FPGAs. In this paper, we are proposing architectural modifications to regular FPGAs to form a domain-specific hybrid FPGA for AI/ML applications.

III. Proposed Architecture: Hamamu

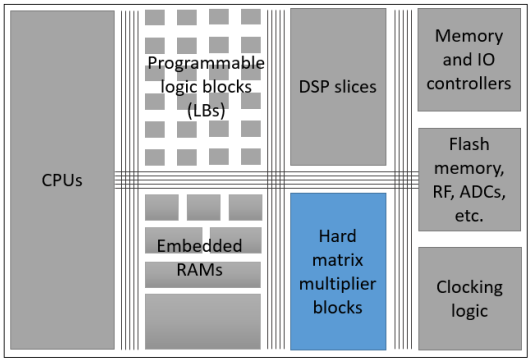


Fig. 1: Block diagram representation of the Hamamu FPGA architecture. Hamamu contains hard matrix multiplier blocks in addition to LBs, RAMs, DSP slices, CPUs, memory/IO controllers, etc. in the FPGA. Note that this diagram is not micro-architectural and is not to scale.

Figure 1 shows a high level block diagram of an FPGA based on our proposal, Hamamu. Hamamu contains some hardened matric multiplier blocks in addition to the usual components of an FPGA such as programmable logic blocks, block RAMs, DSP slices, programmable interconnect, etc. A part of the silicon area available for logic blocks or DSP slices is converted to the hardened matrix multiplier blocks. We explain the various aspects of the proposal in the sections below.

A. Hard matrix multipliers as building blocks

A matrix multiplier designed using soft logic (LBs and interconnect on an FPGA) is slow and area-inefficient. DSP slices can be used to design matrix multipliers that are faster than those designed with LBs. A DSP slice usually contains a multiplier or a MAC. Matrix multiplication requires many MAC operations. Therefore, multiple DSP slices have to communicate using the FPGA interconnect resources to make even a small matrix multiplier. This makes such matrix multipliers slower compared to dedicated ASIC matrix multipliers.

In addition to the core multiply-and-accumulate operation, a matrix multiplier design has some control logic as well that orchestrates data movement from/to the memories. Designing this logic using LBs and FPGA interconnect also slows down the overall operation of the design.

We propose adding hard matrix multiplier blocks to existing FPGAs. This has the following benefits:

- It increases the compute density of the FPGA fabric, providing more floating-point operations per unit area (FLOPs/mm^2).
- It reduces the overall silicon area required to implement a given operation or a layer on the FPGA.
- It can lead to designs with faster frequencies because of the reduced dependence on LBs, DSP slices and FPGA interconnect.

B. Implementation of the hard matrix multiplier building block

Multiple implementations of a matrix multiplier are possible. **Systolic arrays** [24] have been deployed for performing matrix multiplications in many designs [6] [11]. These architectures have many interesting properties, including reusing a piece of data multiple times and never having to read it again, making them very efficient for compute-intensive tasks like matrix multiplication. A systolic array based implementation of a matrix multiplier comprises of 3 pieces of logic:

- Processing elements (PEs) arranged in the form of an array or matrix
- An input data setup circuit that fetches the input data from the producer or memory and provides the data to the MACs at the right time
- An output interface circuit that writes the data to the consumer or memory

There are multiple types of systolic array implementations as well. For our experiments, we used an adaptation of Design R1 [24] described in [25]. The elements of one matrix move from top to bottom and the elements of the second matrix move from left to right. The result *stays* in the respective PE until its computation is done, before shifting out. We assume input matrix A is stored in RAM in column-major order and input matrix B is stored in RAM in row-major order.

We use the notation that a MxNxK matrix multiplier multiplies a MxK matrix (matrix A) with a KxN matrix (matrix B) to produce a MxN matrix (matrix C). For a 4x4x4 matmul, our implementation reads 8 input elements per clock cycle (4 elements of matrix A and 4 elements of matrix B). There is a grid of 16 PEs, each consisting of a pipelined MAC unit. The input elements flow through the grid and accumulated sums stay in the PEs. The output elements are shifted out along the 4 rows when accumulations have completed and written to a RAM.

C. Size of the hard matrix multiplier building block

As mentioned in [5], there are area, speed and power tradeoffs when comparing building blocks of different sizes. For this work, we considered matmuls with M=N=K and each of M,N,K were a power-of-2. We considered 2x2x2, 4x4x4, 8x8x8 and 16x16x16.

Designing large matrix multipliers using smaller matmuls means using the FPGA interconnect (switchboxes and connection boxes) for any communication between the matmuls, which adds additional delays and slows down the overall frequency of operation. Larger matmuls (e.g. 16x16x16 matmul) lead to higher speed, less area and reduced power consumption for a given design, but they also lead to more routing area per block, increased channel width and increased average net length [21].

The problem of under-utilization or fragmentation happens when we have a big matrix multiplier block (e.g. 16x16x16) available, but a smaller problem/design size (e.g. 12x12x12). This also happens when we have a larger problem size (e.g. 14x14x14), but matrix multipliers that are available are smaller and do not evenly divide the edges of the problem size (e.g. 8x8x8). Providing smaller sized matrix multipliers on an FPGA means having less under-utilization and fragmentation problems, compared to providing larger sized matrix multipliers.

D. Composing building blocks to make bigger matrix multipliers

State-of-the-art neural networks require matrix multiplications of varying sizes, most of them being very large. Large matrix multiplications can be done by composing smaller matmul blocks. We discuss two ways of composing matrix multipliers below:

- **Parallel composition** involves using building block matmuls for each slice of the larger matrix multiplication problem. A 8x8x8 matrix multiplication using this scheme requires 8 4x4x4 matmuls and 4 matrix additions. This scheme produces the result in fewer cycles (4M-2+P+1, where M is the size of the matmul used and P is the number of pipeline stages in the MAC), but needs more hardware resources.
- **Systolic composition** involves connecting building block matmuls in a 2D arrangement systolically, just like individual PEs inside a matmul connect. A 8x8x8 matrix multiplication using this scheme requires 4 4x4x4 matmuls (see Figure 2). This scheme takes more number of cycles than parallel composition to produce the result (4*N-2+P, where N is the size of the matrices being multiplied and P is the number of pipeline stages in the MAC), but it uses fewer hardware resources. This scheme results in a

high compute efficiency because only the matmul blocks along the top and left edges of the larger multiplier fetch data from memory, while the other matmul blocks receive inputs from their neighboring matmuls.

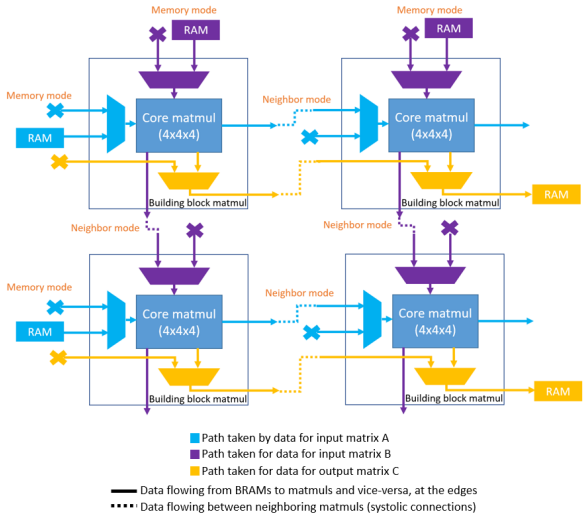


Fig. 2: A 8x8x8 matrix multiplication using systolic composition is done by connecting 4 4x4x4 systolic matmuls. The control logic in each building block matmul is designed to enable neighboring matmuls to connect either in a neighbor mode or in a memory mode, allowing for systolic composition.

A systolic composition can be done using two methods. In the first method, the connection between the smaller matrix multipliers is done using soft interconnect (connection boxes and switch boxes). In the second method, the connection between the smaller matrix multipliers is done using **programmable direct interconnect**. This interconnect is an additional element of our proposed architecture. This interconnect is provided from each building block matrix multiplier to four neighboring building block matrix multipliers - left, right, top and bottom. These connections are configured at the FPGA configuration time. From an implementation perspective, these are basically wire segments with longer length and one switching element (a pass transistor or transmission gate) controlled by a single-bit SRAM cell. Most modern FPGAs provide direct interconnect between DSP slices only in the vertical direction.

E. Placement of hard matrix multiplier building blocks

We explore several placement options for these blocks alongside the other blocks on the FPGA. The **Columnar** placement method (shown in Figure 3 (a)) is inspired by most commercial FPGAs. In this case, columns of matmuls are spread evenly between columns of other blocks like LBs and RAMs. This type of placement is used to analyze the performance of a dedicated ML-specialized FPGA fabric. The **Surround** placement method (shown in Figure 3 (b)) is the recommended method from [21]. The matmuls are placed such that they are surrounded by RAMs and LBs. The **Hybrid** placement method (shown in Figure 3 (c)) is basically a Columnar placement with columns containing DSP slices as well. This placement strategy is to study the impact of just *adding* matmuls to existing commercial FPGA architectures, whereas the other 2 strategies *replace* DSP slices in existing FPGA architecture with matmuls.

IV. Experimental Methodology

A. Tools and parameters

We used the following tools to explore the architectures proposed in this paper:

- VTR 7.0 for FPGA architecture exploration [26]
- Synopsys VCS 2017.12 for Verilog simulations [27]
- Synopsys Design Compiler 2017.09 for ASIC synthesis [28]

VTR is an academic tool that enables exploration of FPGA architectures. VTR takes two inputs. The first input is an architecture description file, where the information of an FPGA's building blocks and interconnect resources is provided. The second input is a benchmark in form of a Verilog design that we intend to overlay onto the FPGA. VTR synthesizes and implements the provided design on to the FPGA with the provided architecture, and generates area and timing reports.

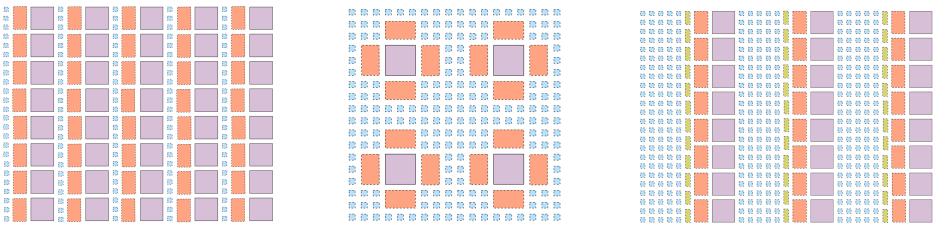
For all the experiments conducted for this research, here are some parameters:

- We performed two sets of experiments - one with 8-bit fixed point (int8) data and another with IEEE Half-precision Floating Point (fp16). These are the two most common precisions used in deep neural networks today.
- We ran VTR in its default mode, in which the tool finds the minimum required value of W (routing channel width) for the given design, and then routes the design again at 1.3x the minimum routing channel width. This is a common practice in research and industry.
- The designs overlayed onto the FPGAs were matrix multiplier designs, with sizes ranging from 4x4x4 to 64x64x64. The designs were hand coded and building block multiplier blocks manually instantiated and connected. The designs included RAMs to store the input and output matrices and logic to interface with the RAMs.

B. FPGA architectures for evaluation

For **the FPGA with DSP slices** (baseline FPGA architecture for our experiments), we created an approximation of the Intel Stratix 10 FPGA architecture (14nm) [29]. We used a Stratix IV architecture (40nm) available with VTR and modified it to obtain this architecture. The delays and areas were modified by scaling based on equations present in [30]. Here are the important features of this architecture:

- LBs: N=10, fracturable 6-LUT with 3-input LUT mode, 5-input LUT mode, 6-input LUT mode and arithmetic mode
- RAMs: 20 Kbits memory that can operate in multiple depths/widths in single port and dual port modes
- Routing architecture: L = 4, Fc_in = 0.15, Fc_out = 0.1, Wilton switches with Fs = 3



(a) Columnar **(b)** Surround **(c)** Hybrid

Fig. 3: Various placement strategies. Three types of blocks (Purple=Matmul, Red=RAM, Blue=LB, Yellow=DSP). The total number of matmuls in the FPGA was kept the same in our experiments, for each placement strategy. Source: VTR.

The DSP slice in this architecture was a custom designed unit that has 3 modes: a multiplier mode, an adder mode and a MAC mode. It supports either int8 operations or fp16 operations. The MAC operation in the DSP slice is deeply pipelined (3 stages for int8 and 8 stages for fp16).

For **the FPGA with hard matrix multiplier blocks**, we used the architecture mentioned above, but replaced the DSP slices with matmul blocks. The same deeply pipelined MAC architecture designed for the DSP slice was used at the heart of the matmul design. We designed matmul blocks of various sizes (2x2x2, 4x4x4, 8x8x8 and 16x16x16) and synthesized them using the FreePDK45 [31] library. The timing and area numbers obtained were then scaled and annotated into the 14nm FPGA architecture. Note that we assumed 15% area overhead of place and route [20]. A 4x4x4 matmul was about 12-15 times larger than a DSP slice. We defined the geometry of the matmuls blocks to be square because [21] finds that a square aspect ratio is the most efficient. We defined the matmuls to have switch boxes inside them because that leads to better routability [22]. Moreover, we defined them to have pins evenly distributed along the perimeter [21].

Various architectures were created for different placement strategies, with different building block sizes and with/without direct programmable interconnect. VTR's FPGA architecture specification language supports specifying direct (non-programmable) inter-block connections. While not exactly the same, we used this feature to model the programmable direct interconnections between neighboring matmuls.

To ensure fair comparisons, the total number of MACs in the baseline FPGA with DSP slices was kept the same as that in the FPGA with matmuls. The designs overlayed on the baseline and proposed FPGA architectures were identical.

C. Analyzing end-to-end benefits for neural networks

Various layers in today's neural networks can be classified into two categories:

1) **GEMM layers**. These layers perform matrix multiplication. Fully connected layers and convolution layers are expressed as GEMMs. Most commonly, these layers are compute bound, especially with larger batch sizes. These are the layers that our proposal tries to improve.
2) **Non-GEMM layers**. These layers include other tasks such as batch normalization, element wise additions, activations (ReLU, Sigmoid, etc). In general, these tasks are memory bound. The proposal in this paper does not affect these layers.

We collected attributes for all layers (GEMM dimensions, number of bytes read, number of bytes written, etc) in 4 MLPerf [4] benchmark networks - Resnet50, GNMT, Transformer and Minigo. We wrote an analytical model that took the attributes for various layers and calculated the number of cycles taken by each layer on the proposed FPGA and on the baseline FPGA, assuming that the computation time for non-GEMM layers can be hidden behind the memory access time (eg. by using pipelining) and that the memory access time for GEMM layers can be hidden behind the computation time (eg. by using double buffering). Using these cycle times, we calculated the overall speedup provided by the proposed FPGA for running a neural network.

V. Results

A. Benefits of hard matrix multiplier blocks

Through a simple experiment, we observe that a 4x4x4 matrix multiplier could be designed at a frequency of 1143 MHz and an area of 1.55E07 MWTAs (Minimum Width Transistor Area) on an FPGA with DSP slices and could be designed at a frequency of 1932 MHz and an area of 4.69E06 MWTAs on an ASIC.

Figure 4 shows the clock frequency achieved when 16x16x16 matrix multiplier was implemented using an FPGA with DSP slices, FPGA with 4x4x4 matmuls (Hybrid placement) and an ASIC. From the figure, we can observe that as we move from left to right, the clock frequency increases and the total area reduces. A reduction in power consumption proportional to the area reduction can be expected.

Table I shows the results obtained when larger matrix multiplier designs are created. We show how the frequency and the area change for an FPGA with DSP slices vs. an FPGA with 4x4x4 matmuls (with Columnar placement strategy). For the 64x64x64 design, we can see a reduction of ˜7.3x in total area and a speedup of ˜3.6x in clock frequency when 4x4x4 matmuls are used, compared to the baseline FPGA.

B. Evaluating different sizes of matrix multiplier building blocks

We evaluated different matmul sizes to identify the best size for an FPGA. We considered two cases. The first case was a design with fragmentation issues. A 35x35x35 matrix

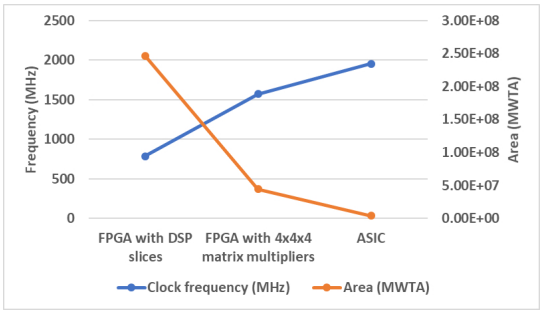


Fig. 4: Frequency and area of a 16x16x16 design with decreasing granularity of the compute element. An FPGA with hard matmuls can close the gap between ASICs and FPGAs.

Design size	FPGA with matmuls		FPGA with DSP slices	
	Clk freq (MHz))	Area (MWTA)	Clk freq (MHz)	Area (MWTA)
4x4	1932.05	4.29E+06	1143.01	1.55E+07
8x8	1927.28	1.19E+07	1072.25	6.58E+07
16x16	1685.18	3.84E+07	788.78	2.47E+08
24x24	1689.45	7.70E+07	658.71	5.35E+08
32x32	1613.75	1.34E+08	462.90	9.72E+08
64x64	1213.72	4.92E+08	333.56	3.62E+09

TABLE I: Different design sizes using an FPGA with DSP slices and an FPGA with 4x4x4 matmul blocks

multiplier was designed using various building block sizes. Table II shows the results from these experiments. Because of fragmentation effects, more time was consumed when larger matmuls are used. The utilization of the matmuls is much higher with smaller building blocks.

Matmul	Freq (MHz)	Cycles	Time (us)	Utilization
2x2x2	1715.27	150	0.0875	0.94
4x4x4	1848.43	150	0.0812	0.94
8x8x8	1872.66	166	0.0886	0.76
16x16x16	1926.12	198	0.1028	0.53

TABLE II: A matrix multiplier with high fragmentation problems (35x35x35) designed using different matmul sizes

For the second case, a design size without fragmentation issues was considered. Figure 5 plots the area-delay product for a 32x32x32 matrix multiplier design using different granularities of the main compute element. We find that the area-delay product significantly improves as we move from using DSP slices to 2x2x2 matmuls. It further improves as the building block size increases, because more logic and interconnect is getting hardened. With larger building block sizes, we observed larger net length, more wire segments per net and also a higher channel width.

Considering both fragmentation effects and the area delay product, we recommend providing 4x4x4 matmuls.

C. Benefits of systolic composition and direct interconnect

Figure 6 compares the time taken and area required to compose smaller matrix multiplier blocks to form larger matrix multipliers by using different approaches mentioned in Section III-D. In this experiment, the final design was a 16x16x16 multiplier and the size of the matmul blocks was 4x4x4. Parallel composition requires more than 4 times the area and systolic composition requires 3x cycles.

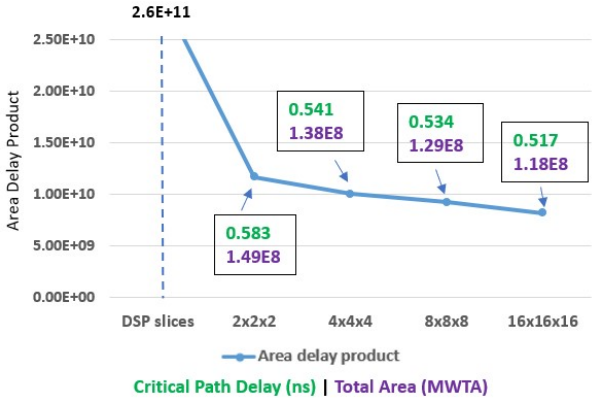


Fig. 5: A matrix multiplier with no fragmentation problems (32x32x32) designed using varying granularity of the compute element. Number of clock cycles consumed is 134 in all cases.

Also shown in Figure 6 are the benefits observed by using direct programmable interconnect. The speedup is higher when the spacing between matmuls in the FPGA layout is higher. The figure shows 15% benefit in the case of hybrid FPGA placement. Another important result we observed was a reduction in the routing area when direct programmable interconnect was introduced.

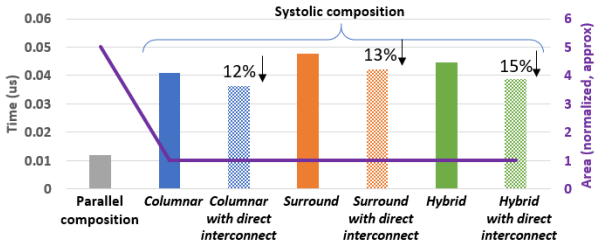


Fig. 6: Parallel composition takes smaller time, but a lot of extra hardware, compared to Systolic composition. The benefit of direct programmable interconnect is also shown for each placement strategy. (Columns = Time, Line = Area)

D. Comparison of various placement strategies of matmul blocks

Figure 7 compares different placement strategies mentioned in Section III-E. The building block matmul size used for these experiments was 4x4x4. The Surround placement (Figure 3 (b)) and Hybrid placement (Figure 3 (c)) result in large total area consumption for a given design size because the matmuls are placed far apart from each other. Columnar placement yields the lowest total area of the 3 placements. Columnar placement (Figure 3 (a)) results in the highest channel widths because matmuls, which are larger building blocks compared to LBs and DSPs, are close to each other causing higher routing congestion.

The clock frequencies achieved using Columnar placement are the highest, because the benefit seen with Columnar placement is because of a matmul-heavy resource mix, in addition to hardening of the compute and interconnect within the matmuls. The ratio of number of LBs to matmuls in Columnar placement is low and so the FPGA is not going

to be as versatile. The Surround and Hybrid placements have a more generic resource mix. So, the decision on which type of placement to deploy depends on the intended usecase of the FPGA.

E. End-to-end benefits for state-of-the-art neural networks

Using the analytical model described in Section IV-C, state-of-the-art networks from MLPerf [4] were evaluated to measure the speedup of implementing them on an FPGA with hard matmuls vs. the baseline FPGA. We considered two batch sizes (1 and 128), we only considered the forward pass, and the building block for the proposed FPGA was 4x4x4 matmul. For GEMM layers, we assumed the FPGA has enough resources for 4 64x64x64 matrix multiplications to happen in parallel. For non-GEMM layers, we assumed off-chip DRAM bandwith of 1.5 TB/s. Figure 8 shows the speedup. On an average, a speedup of ˜3.9x was obtained by using an FPGA with hard matrix multiplier blocks with Columnar placement and with direct programmable interconnect between neighboring matmuls. The benefits reduce to ˜2.6x for Surround placement and ˜2.8x for Hybrid placement.

F. Results for experiments with 8-bit fixed point precision

We performed experiments using int8 precision as well. The trends obtained with int8 precision are very similar to those obtained with fp16 precision. A 16x16x16 matrix multiplier design using an FPGA with 4x4x4 matmuls (Hybrid placement) was 2.5x faster than the same design using an FPGA with DSP slices, and took 8.3x smaller area. Comparing various matmul sizes, we observed 4x4x4 matmul outperforming other matmul sizes, similar to fp16. Direct programmable interconnect provided a benefit of up to 15%.

VI. Discussion

Target of our research: Admittedly, an FPGA with matmuls will still be less performant than ASICs like the Google TPU. The intent of this proposal is to improve the performance of FPGA based solutions. In the process of improving the performance, this proposal makes the FPGA less flexible, thereby making the FPGA less attractive for applications that do not require matrix multiplications. However, a matmul-heavy FPGA fabric could be deployed as a part of bigger FPGA, the rest of which can have general programmable logic, or ML-specific FPGA variants with matmuls could be created.

Speeds are much faster than Stratix-10: It can be seen that the frequencies achieved by our designs are significantly high (˜1900 MHz) compared to the frequency at which Stratix-10 FPGA's DSP slices can run (750-1000 MHz). There are several reasons for this. We only support one data precision at a time (int8 or fp16), unlike DSP slices in Stratix-10, which support multiple precisions including larger precisions like fp32. Also, our designs are more deeply pipelined (8 stages for fp16) compared to the designs in Stratix-10 (˜5 pipeline stages for fp32).

Benefits are pessimistic: The architecture for both comparables (proposed FPGA with matmuls and baseline FPGA with DSP slices) was kept the same to ensure apples-to-apples comparison. The same pipelined MAC architecture designed for the DSP slice was used in the matmul design. Even with that, the benefits we have calculated are actually pessimistic. That's because the DSP slice we have used performs much better than DSP slices on current FPGAs, because it does not have muxes to support many modes and precisions. In other words, our baseline is faster than what is commercially available. Also, many commonly used techniques (hardware or software) to minimize reading and writing from DRAM (such as keeping as much data as possible in on-chip RAMs and reusing it, or fusing memory bound operations, e.g. ReLU, with compute bound operations, e.g. GEMM) will reduce time taken by memory bound layers and actually help in amplifying the overall benefit from our approach.

Future work: Although this research has shown promising results, we have identified some future work. Currently, we designed separate matmuls for int8 and fp16 precisions. We plan to design matmuls which can support multiple precisions, but use the least area. We also plan to study the impact of adding matmuls to the FPGA on non-ML applications, by using benchmarks available with VTR. We will look into how this impact can be reduced. Another aspect of future work is to analyze the routing/crossbar/interconnect requirements inside the matmul block, as is done in commercial FPGA DSP slices and LBs.

VII. Conclusion

In this paper, we propose adding hard matrix multiplier blocks to the architecture of existing FPGAs to reduce the gap between FPGAs and ASICs. We recommend adding systolic array based 4x4x4 hard matrix multipliers as building blocks to the fabric of an FPGA. Larger matrix multipliers can be composed by systolically connecting these building block matrix multipliers by using direct programmable interconnect provided between neighboring matrix multiplier building blocks.

Adding matmuls to the FPGA fabric increases the compute density of the FPGA fabric, providing more floating-point operations per unit area (FLOPs/mm^2). We simulate a 64x64x64 matrix multiplier design on a proposed FPGA architecture with 4x4x4 hard matrix multiplier blocks using Columnar placement with direct programmable interconnect. Experimental results show a clock frequency speedup of ˜4.1x and an area improvement of ˜7.3x on this FPGA, compared with implementing the same design on a DSP-heavy Stratix 10-like FPGA architecture with equal number of MACs, same MAC architecture and high DSP:LB ratio. The changes proposed in this work lead to a highly performant domain specific FPGA, and with the abundance of AI/ML applications where FPGAs can be deployed, implementing these changes is a cost worth paying.

References

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

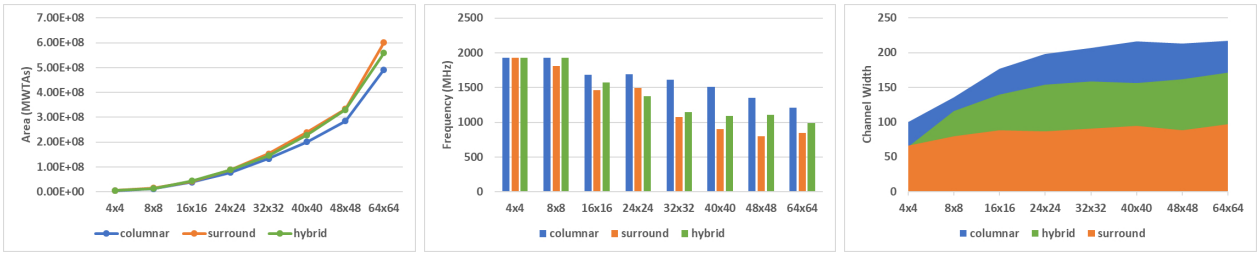


Fig. 7: Comparing different placement strategies. Columnar provides the most benefit, but is very ML-specific. Surround and Hybrid have lower benefits, but have more versatile resource mix.

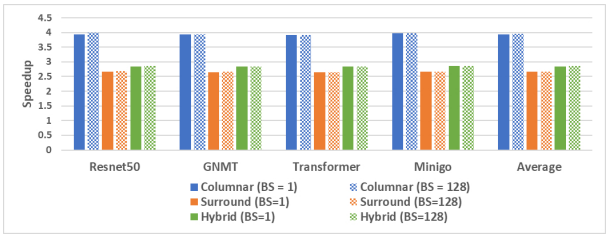


Fig. 8: Overall speedups for various networks. BS = Batch Size. Results include the benefits from direct programmable interconnect.

[2] Intel. (2019) Intel Agilex FPGAs and SOCs. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/fpga/agilex.html

[3] E. Nurvitadhi, S. Shumarayev, A. Dasu, J. Cook, A. Mishra, D. Marr, K. Nealis, P. Colangelo, A. Ling, D. Capalija, and U. Aydonat, "In-Package Domain-Specific ASICs for Intel® Stratix® 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC," 02 2018, pp. 287–287.

[4] mlperf.org. (2018) Mlperf. [Online]. Available: http://www.mlperf.org

[5] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends® in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2008. [Online]. Available: http://dx.doi.org/10.1561/1000000005

[6] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," *CoRR*, vol. abs/1704.04760, 2017. [Online]. Available: http://arxiv.org/abs/1704.04760

[7] NVIDIA. (2017) NVIDIA TESLA V100 GPU ARCHITECTURE. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[8] Y. Chen *et al.*, "DaDianNao: A Machine-Learning Supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 609–622. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.58

[9] J. Fowers *et al.*, "A Configurable Cloud-scale DNN Processor for Real-time AI," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 1–14. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00012

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

[11] M. S. Abdelfattah *et al.*, "DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration," *CoRR*, vol. abs/1807.06434, 2018. [Online]. Available: http://arxiv.org/abs/1807.06434

[12] Flex-Logix. (2019) Flex-Logix EFLX eFPGA. [Online]. Available: https://flex-logix.com/wp-content/uploads/2019/09/2019-09-EFLX-4-page-Overview-TGF.pdf

[13] ——. (2019) Flex-Logix nnMAX Inference Acceleration Architecture. [Online]. Available: https://flex-logix.com/wp-content/uploads/2019/09/2019-09-nnMAX-4-page-Overview.pdf

[14] Achronix. (2019) Speedster7t FPGAs. [Online]. Available: https://www.achronix.com/product/speedster7t/

[15] A. Boutros, M. Eldafrawy, S. Yazdanshenas, and V. Betz, "Math Doesn't Have to be Hard: Logic Block Architectures to Enhance Low-Precision Multiply-Accumulate on FPGAs," 02 2019, pp. 94–103.

[16] A. Boutros, S. Yazdanshenas, and V. Betz, "Embracing diversity: Enhanced dsp blocks for low-precision deep learning on fpgas," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 35–357.

[17] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong, "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)*, 2019, pp. 35–44.

[18] E. Nurvitadhi *et al.*, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021740

[19] G. Lacey, G. W. Taylor, and S. Areibi, "Deep Learning on FPGAs: Past, Present, and Future," *CoRR*, vol. abs/1602.04283, 2016. [Online]. Available: http://arxiv.org/abs/1602.04283

[20] C. Ho, C. Yu, P. Leong, W. Luk, and S. Wilton, "Domain-Specific Hybrid FPGA: Architecture and Floating Point Applications," 09 2007, pp. 196 – 201.

[21] C. W. Yu, J. Lamoureux, S. J. E. Wilton, P. H. W. Leong, and W. Luk, "The Coarse-Grained / Fine-Grained Logic Interface in FPGAs with Embedded Floating-Point Arithmetic Units," in *2008 4th Southern Conference on Programmable Logic*, March 2008, pp. 63–68.

[22] C. Yu, W. Luk, S. J.E. Wilton, and P. Leong, "Routing optimization for hybrid FPGAs," 01 2010, pp. 419 – 422.

[23] P. A. Jamieson and J. Rose, "Enhancing the Area Efficiency of FPGAs With Hard Circuits Using Shadow Clusters," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 12, pp. 1696–1709, Dec 2010.

[24] H. T. Kung, "Why Systolic Architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982. [Online]. Available: https://doi.org/10.1109/MC.1982.1653825

[25] R. M. Keller. (2011) Systolic Arrays and Algorithms. [Online]. Available: https://www.cs.hmc.edu/courses/2011/spring/cs156/Systolic.pdf

[26] J. Luu *et al.*, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, pp. 6:1–6:30, June 2014.

[27] Synopsys. (2018) Synopsys VCS. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html

[28] ——. (2018) Synopsys Design Compiler. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[29] Intel. (2015) Stratix 10 fpga features. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html

[30] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm," *Integration, the VLSI Journal*, vol. 58, pp. 74–81, 2017, http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIintegration.TechScale/.

[31] NCSU. (2018) FreePDK45. [Online]. Available: https://www.eda.ncsu.edu/wiki/FreePDK45:Contents