Efficient Approaches for GEMM Acceleration on Leading AI-Optimized FPGAs

Endri Taka[†§], Dimitrios Gourounas[†§], Andreas Gerstlauer[†], Diana Marculescu[†], Aman Arora[‡]
[†]*The University of Texas at Austin, USA*, [‡]*Arizona State University, USA*
{endri.taka, dimitrisgrn, gerstl, dianam}@utexas.edu, aman.kbm@asu.edu

***Abstract*—FPGAs are a promising platform for accelerating Deep Learning (DL) applications, due to their high performance, low power consumption, and reconfigurability. Recently, the leading FPGA vendors have enhanced their architectures to more efficiently support the computational demands of DL workloads. However, the two most prominent AI-optimized FPGAs, i.e., AMD/Xilinx Versal ACAP and Intel Stratix 10 NX, employ significantly different architectural approaches. This paper presents novel systematic frameworks to optimize the performance of General Matrix Multiplication (GEMM), a fundamental operation in DL workloads, by exploiting the unique and distinct architectural characteristics of each FPGA. Our evaluation on GEMM workloads for int8 precision shows up to 77 and 68 TOPs (int8) throughput, with up to 0.94 and 1.35 TOPs/W energy efficiency for Versal VC1902 and Stratix 10 NX, respectively. This work provides insights and guidelines for optimizing GEMM-based applications on both platforms, while also delving into their programmability trade-offs and associated challenges.**

***Index Terms*—Versal, Stratix, FPGA, AI Engine, AI Tensor Blocks, ACAP, GEMM, Hardware Acceleration, Deep Learning**

I. Introduction

The explosion of computational demands in Deep Learning (DL) workloads [1], [2], has resulted in the emergence of AI-optimized hardware solutions, including GPUs [3]–[5] and ASICs [6]–[9]. In addition, several AI-optimized FPGA solutions have also been proposed, both in industry [10]–[14] and academia [15]–[20]. The two major FPGA vendors have adopted different directions in optimizing their FPGAs for DL. AMD/Xilinx introduced the Versal Adaptive Compute Acceleration Platform (ACAP) [21], [22], comprising the novel AI Engine (AIE), along with reconfigurable logic (FPGA) and scalar processors (CPUs). In contrast, Intel released the Stratix 10 NX [10], maintaining the existing FPGA architecture, but replacing legacy DSP blocks with new AI Tensor Blocks (TBs). The AIE is an out-of-fabric solution consisting of programmable vector processors that operate at high frequency. In contrast, TBs are in-fabric blocks comprising multiple dot-product engines, operating at lower FPGA fabric frequencies.

The two hardware platforms employ substantially different architectural attributes to incorporate DL support. In this work, we present systematic methodologies and novel optimization techniques to map General Matrix-Matrix Multiplication (GEMM) workloads on the aforementioned AI-optimized FPGAs, highlighting the distinct architecture-specific design approaches required for each device. We propose frameworks that aim to maximize the throughput and energy efficiency of GEMM on Versal and Stratix FPGAs, leading to maximal resource utilization. This study focuses on GEMM, since it constitutes the core operation in many DL workloads, occupying up to 90% of the total execution time [23], [24].

[§]Authors contributed equally to this work.

TABLE I: Hardware platform characteristics.

	Device	Versal VC1902	Stratix 10 NX 2100
FPGA	Logic Elements/Cells	1968K	2073K
	On-chip Memory	20.5 MB	16.75 MB
	DSP Slices	1968	–
	Tensor Blocks	–	3960
	AIE Cores	400	–
	AIE Memory	12.5 MB	–
	Processing System	ARM A72 + R5F	–
	DRAM Technology	DDR4	HBM2
	Peak DRAM BW	102.4 GB/s	512 GB/s
	Theor. Peak TOPs (int8)*	135 TOPs	143 TOPs
	Peak Power	165 W	125 W
	Process	7nm TSMC	14nm Intel

* Versal's throughput is primarily attributed to the AIE, providing a peak of 128 TOPs at 1.25 GHz. DSPs present only 7 TOPs at 600 MHz, hence are not considered in this work. Stratix 10 NX peak throughput is reported in [26], for operation at 600MHz.

Table I presents the characteristics of the two exemplar devices, where we showcase our proposed methods; the Versal VC1902 and Stratix 10 NX 2100. Both are large chips with roughly equal number of logic elements/cells, and similar on-chip memory capacity. Additionally, both devices have nearly equal *theoretical peak* throughput ($int8$) capabilities, under similar power envelopes (135 *vs.* 143 TOPs, and 165 *vs.* 125 W, for Versal [25] and Stratix [10], respectively).

Besides their distinct architectures, the two devices also present differences in their DRAM technology and the manufacturing nodes (Table I). Versal has 5$\times$ lower bandwidth (BW) than Stratix. Moreover, Versal is manufactured in a 7*nm* TSMC process, while Stratix uses 14*nm* Intel. The main focus of this paper is to provide a comprehensive evaluation of various aspects in GEMM optimization, emphasizing architecture-specific methodologies, which are largely agnostic to DRAM and manufacturing technology. Thus, we perform experiments on designs that operate within on-chip memory. However, to enable a complete and thorough analysis we extensively examine off-chip memory considerations and requirements for both devices. Our main contributions are summarized below:

- For Versal, we leverage the state-of-the-art MaxEVA framework [27], and extend it to incorporate an additional memory hierarchy level utilizing the Versal FPGA's on-

chip resources. We maximize performance via design space exploration (DSE) and analytical modeling, and we propose a novel RAM optimization scheme to overcome severe limitations of Vitis High-Level Synthesis (HLS).
- For Stratix, we develop a novel framework to design, map and optimize a configurable GEMM accelerator by exploiting the device's in-fabric TBs. Our framework involves extensive DSE and analytical modeling to maximize GEMM performance.
- Demonstration of our frameworks on various GEMM workloads for $int8$ precision, showing throughput up to 77 and 68 TOPs with 100% AIE and 91% TB utilization for Versal and Stratix, respectively. We achieve up to 0.94 and 1.35 TOPs/W energy efficiency, with 88% and 94% on-chip memory for Versal and Stratix, respectively.
- We provide notable insights and guidelines for GEMM optimization, programmability aspects, architectural attributes, and limitations on both AI-optimized FPGAs.

II. Related Work

Several prior works leverage the Versal ACAP architecture across multiple application domains. In particular, CHARM [28], [29] automates the process of GEMM acceleration on Versal ACAP. Their experimental results on the VC1902 exhibit higher energy efficiency of up to 7.2× and 1.7× compared to traditional FPGAs (AMD/Xilinx U250) and GPUs (NVIDIA A100), respectively. MaxEVA [27] is another framework that accelerates GEMM on Versal AIE, while achieving up to 2.19× higher throughput and 20.4% higher energy efficiency compared to CHARM. Additional research on Versal focuses on accelerating specific DL workloads, such as Convolutional Neural Networks (CNNs) [30], [31], and Graph Neural Networks (GNNs) [32], [33]. Other works include AIE compilers [34], arbitrary precision integer multiplication [35], as well as acceleration of atmospheric simulations [36], [37]. Considering all prior work, MaxEVA is the state-of-the-art GEMM implementation, although only targeting small matrix sizes that fit within the Versal AIE. In this work, we extend MaxEVA to support arbitrary GEMM sizes, by implementing an additional level of memory hierarchy on Versal's FPGA.

The NX architecture was introduced in [10], including a discussion of the TB operating modes and design trade-offs. This work also presents a TB design used for General Matrix-Vector (GEMV) and GEMM operations. Multiple other works have targeted the Stratix 10 NX for DL. In [38], a GEMV accelerator is mapped on AI TBs and incorporated into an enhanced Brainwave NPU overlay [39]. They show a speedup of up to 3.5× compared to all prior works on FPGA-based acceleration of Recurrent Neural Networks (RNN) [40]–[42], as well as the baseline NPU with legacy DSPs [43]. Additionally, in [44], NX was utilized to enhance the CNN HPIPE accelerator [45]. They demonstrate a 4× speedup over prior FPGA accelerators for CNN. In [46], a method to assemble higher than $int8$ precision multipliers is presented on the NX. Finally, in [47], a speech-generation model is implemented on Stratix 10 NX, substantially outperforming a V100 GPU implementation. In this work, we develop a framework, which includes a detailed, systematic approach for automatically generating a configurable GEMM accelerator on Stratix 10 NX. Furthermore, we perform an extensive DSE and we explore various trade-offs in GEMM design, which are not thoroughly examined in prior work targeting the NX device.

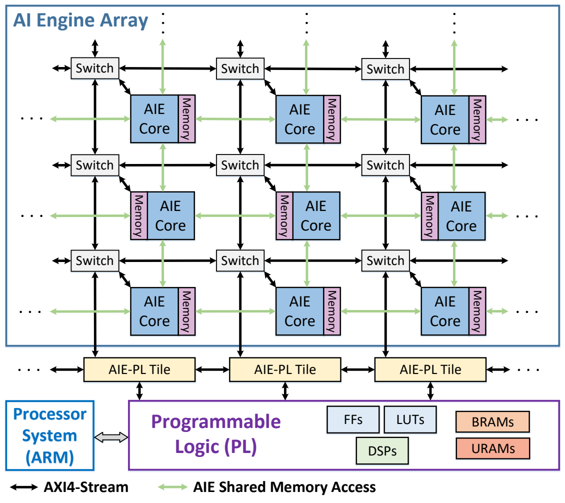


Fig. 1: Versal ACAP architecture.

III. FPGA Architectures Overview

A. Versal ACAP Architecture

The architecture of the Versal ACAP is depicted in Fig. 1. The Versal ACAP comprises the Processor System (PS), the Programmable Logic (PL), as well as the novel AIE array [48]. The PS consists of scalar ARM processors, while the PL includes the traditional FPGA resources, *e.g.*, LUTs, FFs, DSP slices, and on-chip memory resources (BRAMs/URAMs).

The Versal AIE is a 2D array consisting of identical AIE tiles. Each tile includes an AIE core, a memory unit and an interconnect (switch) [49]. The AIE cores are architected as VLIW programmable processors featuring vector (SIMD) units. The AIE array provides three levels of parallelism. First, *instruction-level* parallelism is realized by executing up to 7 instructions every clock cycle (7-way VLIW). Second, *data-level* parallelism is achieved through vector operations, where multiple data can be processed each clock cycle (SIMD). Third, *spatial-level* parallelism is attained via the concurrent execution of multiple AIE cores (up to 400). Communication between different AIE cores is achieved by local memory sharing access for neighboring cores, or by programmable switches for distant cores (Fig. 1). The switches can be configured statically (at compile time) for circuit-switching, or dynamically for packet-switching. Circuit-switching is more efficient, ensuring deterministic latency, as opposed to non-deterministic latency associated with packet-switching [49].

The AIE array communicates efficiently with the PL via the dedicated AIE-PL tiles, located on the last row of the AIE, as shown in Fig. 1. These tiles provide AXI4-Streaming interface with the PL, while also supporting clock domain crossing between the AIE and the PL. The Versal ACAP additionally

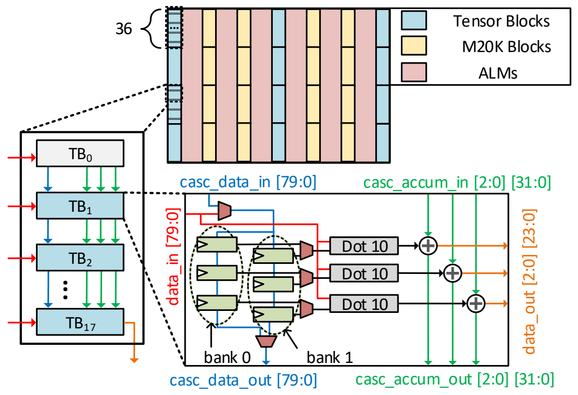


Fig. 2: Architecture of Stratix 10 NX Tensor Blocks.

includes a Network-on-Chip (NoC) (not shown in Fig. 1), to enable flexible communication throughout the entire chip.

An AIE kernel running on a single AIE core can be programmed in high-level C/C++ [50], or low-level SIMD intrinsics [51]. The mapping of multiple kernels on the AIE array is realized through the Adaptive Data Flow (ADF) graph modeling. The nodes in the ADF correspond to AIE kernels and the edges represent connections between them [52]. The PL can be programmed in C/C++ using Vitis HLS [53] or low-level RTL. Finally, AMD/Xilinx provides the Vitis V++ tool [54] to integrate the AIE graph system and the PL kernels.

B. *Stratix 10 NX Architecture*

The Intel Stratix 10 NX 2100 device's PL comprises ALMs, FFs, M20K blocks (Intel's BRAMs, 20Kbit in size [55]) and the AI-optimized TBs. TBs replace the traditional variable-precision DSP blocks (Fig. 2), by dropping many of their legacy, high-precision operating modes and replacing them with scalar, vector, and tensor operating modes for several DL-optimized data types. The TB maintains the interface of the legacy DSP. In this work, we focus on the Tensor $int8$ mode, which supports three 10-element $8{\times}8$ signed dot-product operations and three 32-bit additions for accumulation on partial products. TBs in an FPGA column are *physically* grouped in chains, each consisting of 36 TBs. The TBs within a chain are cascaded using dedicated wires to propagate operands and accumulated products. Neighboring chains are not cascaded. Within a chain, TBs can be *logically* grouped in independent *arrays* of configurable length.

Fig. 2 illustrates an example of an *array* of 18 TBs that lies within a TB chain, as well as a simplified block diagram of the TB operating in the $int8$ mode. Each TB contains two banks of three ping-pong registers for storing operands (*bank* $0, 1$), where each register holds ten 8-bit values. Moreover, it includes three 10-element dot-product engines (*Dot* 10). The first input to each dot-product engine comes from its corresponding register, while the second input is broadcast from the 80-bit wide $data\_in$ port to all three engines. Finally, three independent 32-bit fixed-point adders are responsible for adding the generated dot-products to the cascade input from the previous TB. As shown in Fig. 2, accumulation results are propagated along the TB *array* through cascade connections between a block's $casc\_accum\_out$ port and the following block's $casc\_accum\_in$ port. The latency of the dot-product calculation plus the cascade accumulation is equal to two cycles. The *array's* final TB outputs are exposed through three 24-bit wide $data\_out$ ports.

There are three methods for loading operands in an *array's* TB registers: (i) parallel load mode, (ii) side load mode and (iii) cascade mode. In this work, we focus on the cascade mode, since it leads to less routing congestion, as mentioned in [10]. We refer the reader to [10] for a detailed description of the other modes and their trade-off analysis. In cascade mode, TB_0 (Fig. 2) performs no computation and acts only as a loading port, where operands enter the array through its 80-bit $data\_in$ port. These operands are then propagated to subsequent TBs (TB_1–TB_{17} in Fig. 2) from the $casc\_data\_out$ port of one TB to the next's $casc\_data\_in$ port and eventually stored in the TB registers. This requires three cycles per TB, leading to longer loading latencies as the array grows in size. However, computation can occur concurrently due to the ping-pong registers, allowing to hide the loading latency.

While the TBs support many modes of operation, there is no available support for programming them using Intel HLS [56] or other high-level tools. Hence, the developer is responsible for generating a TB-based design exclusively in RTL.

IV. GEMM Design & Optimization

A. *GEMM Implementation on Versal ACAP*

We leverage the MaxEVA open-source code [57], and extend it to include on-chip buffers in the PL, tiling logic, as well as Load/Store units to communicate with DDR. The PL is designed using Vitis HLS, as extensively used in prior works on Versal to enhance productivity [28], [31], [32], [37]. Optimization of the PL design is attained through analytical modeling for maximization of on-chip data reuse (to reduce DDR BW requirements, as VC1902 has limited BW, Table I). Moreover, this method effectively resolves severe memory over-utilization issues caused by Vitis HLS. In the following sections, we provide a brief overview of the MaxEVA AIE design, and we elaborate on the PL design and optimization.

1) GEMM Multi-Level Tiling Scheme: Fig. 3 illustrates the tiling scheme used in MaxEVA for the AIE, as well as our tiling method on PL. Each AIE core executes a Matrix Multiplication (MatMul) kernel of $M{\times}K{\times}N$ size (*first tiling level*). The parameters X, Y, Z determine the multiple MatMul kernels running on the entire AIE array, as discussed in the next section (*second tiling level*). We incorporate an additional level of buffering in the PL, introducing three new parameters: U, V, W (*third tiling level*). The AIE-specific parameters (X, Y, Z, M, K, N) are optimized by utilizing the MaxEVA framework, while the PL-specific parameters (U, V, W) are optimized using our proposed PL optimization procedure (introduced in Sec. IV-A4). All the aforementioned parameters determine the A, B and C matrix sizes supported out of on-chip memory in the PL, as depicted in Fig. 3. Therefore, we define two GEMM sizes. First, the *compute* GEMM size, *i.e.*, $(X \cdot M) \times (Y \cdot K) \times (Z \cdot N)$, running on the AIE array. Second,

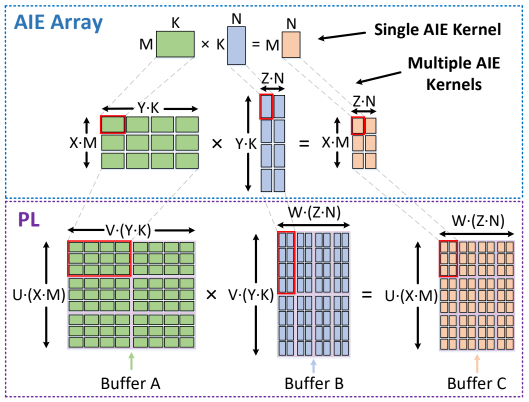


Fig. 3: Multi-level tiling scheme for GEMM on Versal ACAP.

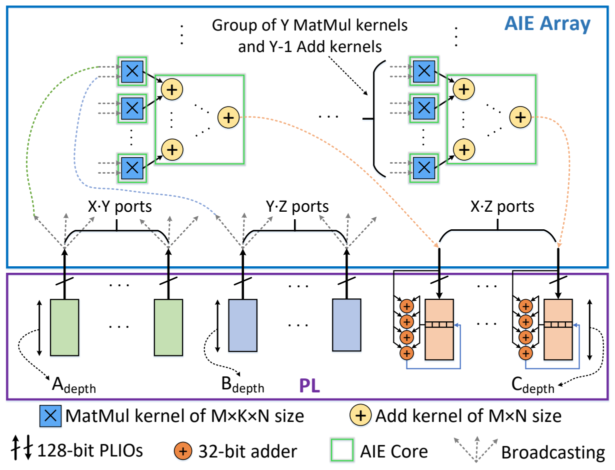


Fig. 4: GEMM accelerator design on Versal AIE and PL.

the *native* buffer size, *i.e.*, $(U \cdot X \cdot M) \times (V \cdot Y \cdot K) \times (W \cdot Z \cdot N)$, of the data stored in the on-chip buffers inside the PL.

2) GEMM Mapping on AIE Array: The limited number of AIE-PL tiles on Versal devices (notably VC1902 has only 39 AIE-PL tiles out of 50 columns in the AIE [58]), is one of the main challenges in GEMM design. To overcome the limited PL Input/Output (PLIO) bottleneck, MaxEVA utilizes the following two techniques [27]. First, the number of input PLIO ports is reduced by broadcasting input data to multiple AIEs. Second, the output PLIO ports are decreased by performing adder tree reduction (via Add kernels) on the AIE. This approach exploits only the most efficient circuit-switching AIE mechanism, as opposed to packet-switching used in [28], [29].

In the upper part of Fig. 4 (AIE array), we present a high-level diagram of MatMul and Add kernels on the AIE. Notice the groups of Y MatMul kernels along with their corresponding adder trees ($Y - 1$ Add kernels). There exist $X \cdot Z$ such groups, all executing in parallel. Each MatMul kernel is mapped to a separate AIE core. All Add kernels of a group (adder tree) are mapped to a single AIE core. A total of $X \cdot Y \cdot Z$ AIE cores execute the MatMul kernels, and $X \cdot Z$ AIE cores execute the Add kernels. Regarding the PLIOs, $X \cdot Y$ and $Y \cdot Z$ input ports are required for matrices A and B, respectively, in addition to $X \cdot Z$ output ports for matrix C.

MaxEVA proposes two AIE kernel placement patterns, referred to as $P1$ and $P2$, to leverage the most efficient local data sharing mechanism of the AIE (Fig. 1), and thus, avoid routing congestion. $P1$ denotes a closely-located placement of each group of $Y = 4$ MatMul kernels and their corresponding adder trees (Fig. 4). Similarly, $P2$ denotes a pattern for $Y = 3$ (refer to [27] for more details on these placement patterns).

3) PL Implementation: The lower part of Fig. 4 (PL) shows a high-level block diagram of the PL design. The A, B and C matrices are stored in on-chip buffers, exploiting the PL BRAM and URAM resources. To provide sufficient bandwidth to/from the AIE, each buffer is partitioned ($HLS\ pragma\ array\_partition$), to exactly match the corresponding AIE PLIO ports. Although not shown in Fig. 4 for simplicity, we employ double-buffering to effectively overlap GEMM computation with external off-chip DDR communication.

We set the PLIO width to 128-bits to ensure rate matching between AIE and PL without performance loss [27], [51]. In addition, while the data type of the A and B buffers is $int8$, all accumulations are performed in 32-bits. Thus, when sending data from the A and B buffers to the AIE, we concatenate 16 8-bit values for each input PLIO to form a 128-bit vector. In contrast, when receiving data from the output PLIOs of the AIE, we pack 4 32-bit values and store them in the C buffer. The *logical* size of the PL buffers is shown in Fig. 3. However, in the *physical* implementation, the buffers have a width of 128-bits to match the PLIO width, and are partitioned into smaller buffers (Fig. 4). The partition factors ($\{A, B, C\}_{part}$) and depths ($\{A, B, C\}_{depth}$) of the buffers are expressed as:

$$A_{part} = 2 \cdot X \cdot Y, \quad A_{depth} = U \cdot V \cdot M \cdot K/16 \qquad (1)$$

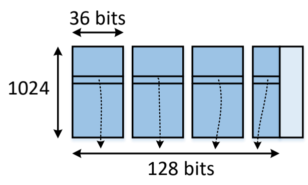
$$B_{part} = 2 \cdot Y \cdot Z, \quad B_{depth} = V \cdot W \cdot K \cdot N/16 \qquad (2)$$

$$C_{part} = 2 \cdot X \cdot Z, \quad C_{depth} = U \cdot W \cdot M \cdot N/4 \qquad (3)$$

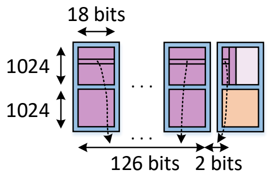
We multiply the partition factor by 2 for double-buffering. We also divide the depth by 16 and 4 to match the 128-bit packing.

Partial results from the AIE are accumulated in the PL to handle the reduction across tiles in the $V \cdot Y \cdot K$ dimension in Fig. 3, or for larger matrices if needed. As depicted in Fig. 4, we implement 4 32-bit adders in soft logic for each partitioned C buffer ($4 \cdot X \cdot Z$ adders in total, all executing in parallel). Each new partial result from every PLIO port (AXI4-Stream interface) is accumulated to its corresponding C buffer address every clock cycle. This PL logic is pipelined ($pragma\ pipeline$) with an Initiation Interval (II) of 1, such that in every clock cycle a new partial result can be accumulated. To this end, one load (for the current PLIO values) and one store operation (for the previous PLIO values) is required for every partitioned C buffer. Thus, we configure the C buffers in *simple* dual-port mode [53], [59] ($pragma\ bind\_storage$). This ensures a stall-free PL implementation that does not introduce any throughput degradation in the entire design.

In contrast, the input buffers A, B are configured in single-port mode, since either only a load (send data to AIE) or store operation (receive data from DDR) is required in each cycle for double-buffering. Moreover, Load/Store units are implemented in the PL (not shown in Fig. 4) to communicate with DDR. Finally, it is important to note that our implementation is



(a) 1K×36 blocks



(b) 2 independent 1K×18 blocks

Fig. 5: BRAM configurations example and proposed modeling.

symmetric in terms of the first and last dimensions in both the *compute* GEMM size and the *native* buffer size (see Sec. IV-A1 for definitions and Sec. V-A2 for evaluation).

4) Memory Optimization Strategy: To maximize the data reuse of the on-chip buffers, we propose an optimization methodology based on analytical modeling. Our model utilizes the multiple configurations of BRAMs and URAMs to identify the optimal values of the U, V, W parameters, as well as the buffer mapping to BRAMs and URAMs. Although Vitis HLS includes the capability to automatically map buffers to memory resources (*impl*=$AUTO$ in $pragma\ bind\_storage$), we found that it fails to find an operational mapping in several cases (Sec. V-A1). This automatic mapping generated by Vitis HLS leads to severe over-utilization of memory resources, and thus, failure to Place and Route (PnR). To the best of our knowledge, this HLS limitation has not been identified in any prior work. Several works have focused on optimizing the logical-to-physical memory mapping during the synthesis/PnR phases [60]–[62], while others [63] propose tools to assist designers at the user-level with automatic memory mapping targeting BRAMs, but not URAMs. Our approach focuses on overcoming this HLS limitation by identifying an optimal mapping to both BRAMs and URAMs, and subsequently guiding the HLS tool according to this mapping (through *impl*=$\{BRAM/URAM\}$ in $pragma\ bind\_storage$).

The inputs of our model include a MaxEVA solution (X, Y, Z, M, K, N parameters), and PL-specific parameters, *i.e.*, BRAM, URAM configurations supported in Versal devices [59] and the available on-chip memory resources. The model produces as outputs the optimal U, V, W parameters, as well as the A, B and C buffer mapping to BRAMs/URAMs.

Besides the optimal parameter finding and mapping to memory resources, our model also estimates the BRAM/URAM utilization with 100% accuracy in all cases (Sec. V-A1). First, we model the BRAM/URAM utilization based on the *depth* of each partitioned buffer (all buffers have 128-bits width and are highly partitioned; Section IV-A3). Fig. 5a shows an example of our modeling when $512 < depth \leq 1K$. In this case, BRAMs are configured as 1K×36 and 4 36K BRAMs are needed to construct the 128-bitwidth buffer. Observe that a portion of the rightmost BRAM in Fig. 5a is not utilized in this situation. To this end, we define BRAM/URAM *efficiency* as the fraction of the *logical* buffer size to the total size of memory blocks used. For instance, when assuming that all 1K entries are used in Fig. 5a, the BRAM *efficiency* is determined by the utilized width, *i.e.*, 128/(36·4) = 88.89%.

In Fig. 5b, we show another example when $1K < depth \leq 2K$. In this case, BRAMs are configured as 2 *independent* 1K×18 blocks. With 7 2K×18 blocks, 126-bits can be mapped. The remaining 2-bits can be efficiently mapped by packing 2K×2-bits on a single 1K×18 block. This is possible since 2K×2-bits can be *logically* viewed as 1K×4-bits, where additional multiplexing logic is needed to determine the corresponding 2-bits, based on the address. Thus, 7.5 36K BRAMs (or 15 18K blocks) are required in total. The remaining 18K BRAM (outlined in orange in Fig. 5b), can be used for other purposes, since it is physically independent.

Similarly, we also model the cases when $depth \leq 512$ (2 36K BRAMs), and $2K < depth \leq 4K$ (15 36K BRAMs) [59]. In addition, since URAMs support one configuration (4K×72), 2 URAMs are required for $depth \leq 4K$. In eq. 4 and 5, we summarize the number of BRAMs (f_B) and URAMs (f_U), respectively, as functions of the buffer depth.

$$f_B(depth) = \begin{cases} 2, & \text{if } depth \leq 512 \\ 4, & \text{if } 512 < depth \leq 1K \\ 7.5, & \text{if } 1K < depth \leq 2K \\ 15, & \text{if } 2K < depth \leq 4K \end{cases} \tag{4}$$

$$f_U(depth) = 2, \quad \text{if } depth \leq 4K \tag{5}$$

Afterwards, we establish a constraint that limits each buffer's depth to 4K (eq. 6), since all buffers are highly partitioned with a relatively small depth. This constraint allows a very high BRAM, URAM *efficiency*, as shown in Sec. V-A2.

$$\{A_{depth},\ B_{depth},\ C_{depth}\} \leq 4K \tag{6}$$

Finally, we impose constraints to ensure that the total number of utilized BRAMs and URAMs does not exceed the device's available resources (B_{36K} and U_{288K}). Eq. 7 and 8 show an example of the constraints for BRAMs and URAMs, respectively. In this example, the buffers A, B have been mapped to BRAMs, while C is mapped to URAMs.

$$A_{part} \cdot f_B(A_{depth}) + B_{part} \cdot f_B(B_{depth}) \leq B_{36K} \tag{7}$$

$$C_{part} \cdot f_U(C_{depth}) \leq U_{288K} \tag{8}$$

Constraints similar to 7, 8 are applied for *all* permutations of mapping buffers A, B and C to BRAMs and URAMs.

The solution of U, V, W and buffer mapping to BRAMs, URAMs can be formulated as an integer programming (IP) optimization problem with the aforementioned constraints. We solve the IP exhaustively by setting the maximization of on-chip data reuse as the objective. The data reuse of all buffers is encapsulated in the product of $U \cdot V \cdot W$. Notice from Fig. 3 that buffers A and B are reused W and U times in GEMM, respectively, while C is reused and updated (during accumulation) V times. In addition, note that maximizing data reuse also leads to maximization of the PL buffer sizes, under the resource constraints. We report, implement and explore the trade-offs of multiple top-ranked solutions in Sec. V-A.

B. GEMM Implementation on Stratix 10 NX

We implement a configurable MatMul accelerator consisting of a control logic and a 2D TB layout on the Stratix 10 NX architecture. The accelerator's TBs operate out of local, on-chip memory consisting of A, B and C buffers (Fig. 2) of size $M'{\cdot}K'$, $K'{\cdot}N'$ and $M'{\cdot}N'$, respectively. The $M'{\times}K'{\times}N'$ MatMul size is defined as the *native* buffer size, similar to Versal. We conduct a DSE to optimize for accelerator throughput and employ analytical modeling to optimize for on-chip data reuse. Below we delineate the TB layout, the dataflow, the memory architecture, optimization strategies and the automatic code generation tool we developed.

1) TB Layout: We utilize groups of TB *arrays* of configurable length that run in parallel and operate in the cascade loading mode. An *array's* first TB (colored white in Fig. 6) serves as its point of entry for A data (Fig. 6), performing no computation. Subsequent TBs in the *array* receive their A data (and store them in their registers) via their *casc_data_in* port and their B data through their *data_in* port (Fig. 2). The latency of loading A blocks is three cycles per TB, but can be hidden when overlapped with dot-product operations. While *bank* 0 (Fig. 2) is being multiplied with a set of B blocks, *bank* 1 can be loaded and vice-versa. An *array's* accumulated outputs are exposed through the *data_out* port of its final TB. We define four architecture parameters for the TB layout:

a) Parameter TB_{len}: Length of a TB *array*, equal to four in Fig. 6. This length can be less than or equal to 36.

b) Parameter K_p: Set of *arrays* that work in parallel across the K' dimension, which we refer to as a *reduction group* (K_p equal to two in Fig. 6). The *data_out* outputs of all the *arrays* in a *reduction group* are fed into its corresponding adder tree, the output of which is accumulated at the C buffer (Fig. 6). These adders are implemented in soft logic.

c) Parameter N_p: Set of *reduction groups* that contain the same A blocks, but get multiplied with different B blocks (equal to two in Fig. 6). N_p allows exploiting parallelism across N'. We refer to this set of *reduction groups* as an *N_p block.*

d) Parameter M_p: Number of *N_p blocks* that allow parallelism across M', equal to three in Fig. 6. Each *N_p block* uses different A blocks, but the same B blocks.

2) Dataflow: Each TB holds a $3{\times}10$ block of A in its registers, represented by a shape in Fig. 6, and it multiplies it with a $10{\times}1$ block of B in each clock cycle. This B block belongs to a set of N'/N_p $10{\times}1$ blocks, represented by a color in the B Buffer (Fig. 6). The TB will sequentially multiply its A block with all B blocks in the colored set. Given enough B blocks in this set (*i.e.*, a large enough N'), the TB's register loading latency can be fully hidden. Different TBs of the same *array* hold separate A blocks (shapes) and process different sets of B blocks (colors). Partial dot-products are propagated along an *array* through cascade connections and added to the next dot-products on-the-fly. This process takes two clock cycles per TB (Sec. III-B). Therefore, a TB starts processing data two cycles after its previous TB. Finally, an *array's* accumulated dot-products are exposed at its last TB.

Different *arrays* in the same *reduction group* hold separate blocks of A (shapes) and process separate sets of B blocks (colors) in parallel, each generating a new output every clock cycle. *Reduction groups* of the same *N_p block* hold the same

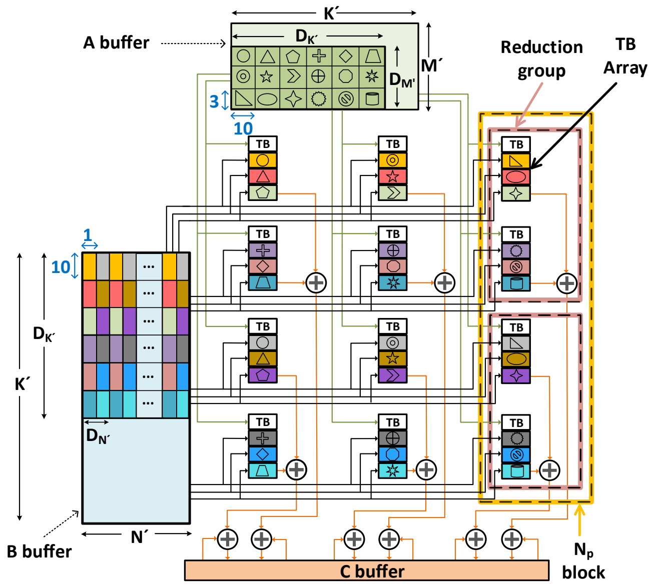


Fig. 6: 2D TB layout and GEMM dataflow on Stratix 10 NX.

A blocks (shapes), but get multiplied with different sets of B blocks (colors) in parallel, allowing parallelism across N'. Finally, different N_p *blocks* contain different sets of A blocks (shapes), but they all get multiplied with the same B blocks (colors) in parallel, allowing parallelism across M'.

The total number of utilized TBs is $TB_{len} \cdot K_p \cdot N_p \cdot M_p$. We define the *compute* GEMM size as $(M_p \cdot 3){\times}([TB_{len}-1] \cdot K_p \cdot 10){\times}(N_p)$. The multiplication by 3 and 10 accounts for the size of the A, B blocks. We subtract by 1 for the wasted TB of each *array* and we multiply by K_p for all *arrays* in a *reduction group*. We refer to this GEMM size as $D_{M'}{\times}D_{K'}{\times}D_{N'}$. This is the size of the blocks the accelerator processes at the lowest memory hierarchy level. Due to tiling over the *compute* GEMM size, M', K' and N' need to be multiples of $D_{M'}$, $D_{K'}$ and $D_{N'}$, respectively. Moreover, N' must be sufficiently large to allow hiding the TB register loading latency.

3) Architectural Considerations: Below we elaborate on architectural considerations related to the four architecture parameters, along with the constraints they introduce.

a) Parameter TB_{len}: As mentioned, the TB *chain* granularity is equal to 36. Hence, as multiple *arrays* may fit in a chain, the *array* length TB_{len} must be a factor of 36 to avoid fragmentation during placement of a design with a large number of TBs. Sound values for TB_{len} are 36, 18, 12 and 9. Low TB_{len} leads to more wasted TBs (first TB in every *array*). Meanwhile, this imcreases flexibility in executing different matrix sizes, as they do not require a high reduction dimension (K'). Large TB_{len} leads to longer latency for loading A, but also results in higher peak throughput. A TB_{len} of 36 has a loading latency of 36·3=108 cycles. However, because each A block is multiplied with a set of B blocks, if this set contains at least 108 blocks, depending on N', this latency can be hidden.

b) Parameter K_p: A TB *array* processes $(TB_{len}{-}1){\cdot}10$ elements across the reduction dimension (K') each cycle. K_p increases the parallelism across K' (*i.e.*, $D_{K'}$), while keeping the signal fan-out low. However, if $D_{K'}$ is large, but K' is small, some TBs in a *reduction group* will be under-utilized.

c) Parameter N_p: In order to utilize more TBs, parallelism across N' and M' is also essential. N_p requires broadcasting A blocks across all *reduction groups* in an N_p *block*, while M_p requires broadcasting B blocks across all N_p *blocks* (Fig. 6). However, each *array* has only one input for A data (at its initial TB), but a considerably larger amount of inputs for B data (equal to $TB_{len}-1$). As a result, M_p increases the fanout of a lot more signals than N_p does, which increases FPGA routing congestion. Finally, a higher N_p leads to a larger minimum required value of N' to hide the TB loading latency.

d) Parameter M_p: M_p enables parallelism across M', but also increases the fan-out of B blocks, which, as mentioned, can lead to significant routing congestion in the FPGA, adversely affecting attainable clock frequency and performance.

4) Memory Architecture: Evidently, when a large number of TBs are used, high BRAM bandwidth is needed to feed them blocks of A and B. We employ a memory architecture that is designed to (i) enable high compute throughput and (ii) to maximize data reuse, which will amortize off-chip BW requirements. In a similar manner to Versal, we maintain separate buffers for A, B and C and we utilize double-buffering. The Load/Store units perform read-only operations on C buffers and write-only operations on A and B buffers. The compute logic performs read and write operations on C and read-only operations on A and B. M20K blocks configured in *simple* dual-port mode are used to implement all buffers. For A and B, the compute and Load/Store units have their own port for reading and writing, respectively. For double-buffering, we double the depth of A and B, and split C to two equal-sized buffers. A more detailed description of the architecture of buffers A, B and C is provided below.

a) Buffers A, B: As illustrated in Fig. 6, a separate A block must be fed to each *array* of a *reduction group* and across different N_p *blocks*. Additionally, each TB in an *array* (except for TB_0), receives separate B blocks every cycle. Similarly, all *arrays* in all *reduction groups* of an N_p *block* require different blocks of B each clock cycle. Also, the bitwidth of the A and B ports is 80-bits. Therefore, the partition factors A_{part}, B_{part} of A, B and the depth A_{depth}, B_{depth} of each smaller partitioned buffer of A, B are:

$$B_{part} = (TB_{len} - 1) \cdot K_p \cdot N_p \tag{9}$$

$$B_{depth} = 2 \cdot K' \cdot N' / (B_{part} \cdot 10) \tag{10}$$

$$A_{part} = M_p \cdot K_p, \quad A_{depth} = 2 \cdot M' \cdot K' / (A_{part} \cdot 10) \tag{11}$$

We multiply by 2 for double-buffering and we also multiply by 10, because 80-bits is 10 bytes. The 80-bit width of these buffers is implemented using M20Ks operating in the 512×40, 1024×20 or 2048×10 configurations [55]. We model the number of M20Ks required to implement each buffer, $f_{M_{80}}$, as a function of the depth. For the A buffer, since A_{depth} is sufficiently large, all M20K configurations lead to equivalent, accurate solutions in our model. However, B_{depth} was always less than 1024. If $depth \leq 512$, two M20Ks in 512×40 mode can be used. If $512 < depth \leq 1024$, both 512×40 and 1024×20 configurations lead to a usage of four M20Ks. As a result, in all our designs $f_{M_{80}}$ is estimated as:

$$f_{M_{80}}(depth) = 2 \cdot \lceil depth/512 \rceil \tag{12}$$

The symbol $\lceil\ \rceil$ denotes rounding up to the next integer.

b) Buffer C: A TB has three *data_out* ports, so *reduction groups* generate three distinct C values each. The partition factor C_{part} and the depth C_{depth} of each smaller buffer is:

$$C_{part} = M_p \cdot N_p \cdot 3 \cdot 2, \quad C_{depth} = M' \cdot N' \cdot 2 / C_{part} \tag{13}$$

We multiply by 2 for double-buffering. The valid M20K configurations for the 32-bitwidth of the C buffers are 2048×8, 1024×16 or 512×32. The number of M20Ks, $f_{M_{32}}$, required for C buffers is also modeled as a function of the depth. Since the depth of the C buffers is sufficiently large, all above M20K configurations led to equivalent, accurate solutions in our model, estimated by:

$$f_{M_{32}}(depth) = \lceil depth/512 \rceil \tag{14}$$

5) Optimization Strategies: Similar to Versal, we formulate the selection of M', K', N' for a given TB configuration as an IP problem and solve it exhaustively to optimize for data reuse, by maximizing the product $M' \cdot K' \cdot N'$. We impose two constraints on the IP. First, the utilized M20Ks must not exceed the device's available M20K blocks (B_{M20K}). Second, N' is set to be sufficiently large to hide TB register loading latency:

$$\begin{aligned} A_{part} \cdot f_{M_{80}}(A_{depth}) + B_{part} \cdot f_{M_{80}}(B_{depth}) \\ + C_{part} \cdot f_{M_{32}}(C_{depth}) \leq B_{M20K} \end{aligned} \tag{15}$$

$$N' \geq TB_{len} \cdot 3 \cdot N_p \tag{16}$$

Since throughput is directly related to operating frequency, we implement optimizations to shorten critical paths. These include replication of control logic and insertion of a configurable number of pipeline stages along the data and address datapaths. Additionally, we conduct an extensive exploration on various TB architecture parameters, to find configurations that maximize frequency, and thus, throughput (Sec. V-B).

In an effort to reduce signal fan-out, we also tried inserting registers between N_p *blocks* in a systolic fashion in order to propagate B blocks, thus minimizing broadcasting. However, our experiments showed no frequency improvements, while greatly increasing ALM usage (up to ∼40%, as opposed to a maximum of ∼19% with the proposed design, see Table IV). Such a large number of soft-logic pipeline registers is undesirable, as it can reduce energy efficiency and also render logic resources unavailable, as mentioned in [10]. Thus, the systolic distribution approach was not considered in our final designs.

6) Automatic RTL Code Generation: We develop a Python-based tool that automatically generates the aforementioned architecture's RTL code, including the control logic, TB layout and memory. The input configurations of the tool are the four architecture parameters, the M', K', N' dimensions, and the number of pipeline stages for both address and data.

V. Evaluation

A. Versal DSE Evaluation

For the AIE, we obtain the two most efficient solutions from MaxEVA. As found in [27], solution $P1$ 13×4×6 ($X{\times}Y{\times}Z$) shows the highest throughput, while the $P2$ 10×3×10 presents the highest energy efficiency. Both use a 32×128×32 ($M{\times}K{\times}N$) single AIE MatMul kernel with 95% throughput efficiency. For the PL, we utilize our model (Sec. IV-A4), which takes as input a MaxEVA solution (X, Y, Z, M, K, N parameters) and produces the optimal values of U, V, W. We perform DSE on the 5 top-ranked $U{\times}V{\times}W$ solutions, for each of the two MaxEVA solutions (10 designs in total).

We use the AMD/Xilinx Vitis 2022.1 version to implement and compile our designs. The PL part is designed using Vitis HLS, while AIE-PL linking is achieved via the V++ compiler. Throughout all experiments, the AIE frequency is set to its maximum value of 1.25 GHz, while the PL frequency ranges from 275–300 MHz, depending on the PL configuration. To calculate the throughput of our designs, we use hardware emulation in Vitis, while power is estimated through the post-implementation Vivado Power Analysis Tool [64].

1) Model Estimation: In Table II, we present 4 top-ranked solutions of our PL optimization procedure (parameters $U{\times}V{\times}W$ and buffer mapping to BRAM/URAM resources). First, we observe that our model estimates the BRAM/URAM utilization with 100% accuracy in all cases. For example, for solution 4×2×4 ($P1$), the model suggests ("Model Est." column) that buffers $\{A, B, C\}$ should be mapped to $\{\mathcal{B}, \mathcal{U}, \mathcal{U}\}$, where $\mathcal{B}$, $\mathcal{U}$ denote BRAM, URAM, respectively. When we guide the HLS tool to use this mapping, both our model and HLS synthesis report identical BRAM/URAM utilization, *i.e.*, 780 (81%) / 408 (88%). Letting the HLS tool instead to automatically map buffers ("HLS AUTO" column), results in a severe over-utilization of URAMs (616 or 133%), without any BRAM usage. Vivado PnR attempts to implement this HLS AUTO solution by mapping the surplus URAMs to BRAMs. However, it generates an error reporting a 119.4% BRAM and 99.8% URAM utilization. A similar result is observed for another solution, *i.e.*, 4×2×4 ($P2$). However, for the other two solutions shown in Table II, HLS AUTO is able to successfully find an efficient mapping. In these cases, both our model and HLS AUTO produce exactly the same BRAM/URAM utilization, *e.g.*, 416 (43%) / 408 (88%), for 2×2×8 ($P1$). We note that HLS AUTO fails to implement 5 of the 10 top solutions (Table III), justifying the necessity of our approach.

2) GEMM Performance: In Table III, we show various metrics for our 10 top solutions. In all solutions, throughput is calculated on their *native* buffer sizes (Sec. IV-A1) Overall, we observe that all designs exhibit high throughput ranging from 75.4–77.01 TOPs. However, to maintain such high throughput, several designs require higher DDR BW compared to the VC1902's BW (102.4 GB/s). We note that we calculate DDR BW as the worst-case of concurrent loads for buffers A, B and stores for C (all as 8-bits due to quantization in DL). With more sophisticated data reuse techniques as in [30], this requirement can be amortized. In this work, we consider the worst-case BW scenario for the sake of generality. Therefore, we only examine designs that stay within the BW of the VC1902 device (highlighted in bold in Table III).

TABLE II: Optimization model estimation for various solutions and comparison with HLS AUTO mapping.

U×V×W	**Model Estimation**			**HLS AUTO**	
(MaxEVA P.)	**{A, B, C}**	**BRAMs**	**URAMs**	**BRAMs**	**URAMs**
4×2×4 (P1)	$\{\mathcal{B},\mathcal{U},\mathcal{U}\}$	780 (81%)	408 (88%)	**0 (0%)**	**616 (133%)**
4×2×4 (P2)	$\{\mathcal{B},\mathcal{B},\mathcal{U}\}$	900 (93%)	400 (86%)	**0 (0%)**	**640 (138%)**
2×2×8 (P1)	$\{\mathcal{B},\mathcal{U},\mathcal{U}\}$	416 (43%)	408 (88%)	416 (43%)	408 (88%)
2×8×2 (P2)	$\{\mathcal{U},\mathcal{U},\mathcal{B}\}$	800 (83%)	240 (52%)	800 (83%)	240 (52%)

From these designs, 2×2×8 ($P1$) shows both the highest throughput, *i.e.*, 76.93 TOPs, and the best energy efficiency, *i.e.*, 0.938 TOPs/W. For all valid solutions, throughput ranges from 75.40–76.93 TOPs, which is 58.9–60.1% of the *theoretical peak* throughput of VC1902, and the same as the state-of-the-art MaxEVA [27]. Additionally, energy efficiency ranges from 0.911–0.938 TOPS/W. In all cases, we notice a very high resource utilization, up to 94% BRAMs, 88% URAMs, and 100% AIE cores, with a small LUT usage of up to 11%. Moreover, we observe high RAM *efficiency* of 75.7–90.2%, as a direct result of our modeling and optimization methodology. Finally, although not shown in Table III for brevity, the swapping of the first and last GEMM dimensions results in equivalent solutions (symmetrical design, see Sec. IV-A3). For instance, solution 2×2×8 ($P1$) can be swapped to 8×2×2 ($P1$), resulting in a *native* buffer size of 1536×1024×832 (*compute* GEMM size becomes 192×512×416 in this case).

B. Stratix DSE Evaluation

We conducted a DSE on 100 designs with different TB architecture parameters to optimize for performance and energy efficiency. The TB utilization of all explored designs is 85%–91%. The *native* buffer sizes (Sec. IV-B) were set based on our IP solver. The RTL (Verilog) code was automatically generated using our Python tool, and afterwards implemented on Intel Quartus 2021.1 (with a patch for Stratix 10 NX support from Intel). We use ModelSim [65] to calculate throughput and the Quartus Power Analyzer to estimate power [66].

Table IV shows the evaluation of the top 10 designs. Configurations are a combination of the 4 TB architecture parameters ($TB_{len} \times K_p \times N_p \times M_p$). Designs are ranked based on their throughput when running GEMM on their *native* buffer sizes. Our solutions achieve high throughput, up to 68 TOPs, which is 47.6% of the *theoretical peak* of the NX 2100 device. We note that our throughput is directly related to our achieved frequencies, which are similar to prior work [10], [38], [44]. The maximum achieved energy efficiency is 1.347 TOPS/W. All designs present a small ALM (12-19%), but very high BRAM (85–94%) and TB (87–91%) utilization. Moreover, they present very high RAM *efficiency* (81.2–90%) and low BW requirements (79.3–92.6 GB/s).

Overall, we notice that the top designs have a TB_{len} of 18, 12 or 9. Designs with a TB_{len} of 36 achieve lower

TABLE III: Evaluation of 10 top-ranked GEMM designs on Versal VC1902. AIE operates at 1.25 GHz.

U×V×W (MaxEVA P.)	Compute GEMM size	Native Buffer size	LUTs	BRAMs	URAMs	AIE cores	PL Fq. (MHz)	Thrpt. (TOPs)	Power (W)	En. Eff. (TOPs/W)	RAM Eff.	BW (GB/s)
2×8×2 (P1)	416×512×192	832×4096×384	85K (9%)	630 (65%)	304 (66%)	390 (98%)	300	77.01	78.6	0.980	88.9%	145.2
2×2×8 (P1)	**416×512×192**	**832×1024×1536**	**91K (10%)**	**422 (44%)**	**408 (88%)**	**390 (98%)**	**290**	**76.93**	**82.0**	**0.938**	**88.9%**	**101.4**
3×2×5 (P1)	**416×512×192**	**1248×1024×960**	**94K (10%)**	**792 (82%)**	**408 (88%)**	**390 (98%)**	**278**	**76.72**	**82.7**	**0.932**	**75.7%**	**100.7**
4×2×4 (P1)	**416×512×192**	**1664×1024×768**	**90K (10%)**	**792 (82%)**	**408 (88%)**	**390 (98%)**	**278**	**76.72**	**82.3**	**0.928**	**81.6%**	**101.9**
2×4×4 (P1)	416×512×192	832×2048×768	97K (11%)	792 (82%)	408 (88%)	390 (98%)	278	76.72	82.8	0.927	62.6%	106.9
2×8×2 (P2)	320×384×320	640×3072×640	92K (10%)	806 (83%)	240 (52%)	400 (100%)	300	76.08	78.3	0.971	88.9%	122.2
2×7×2 (P2)	320×384×320	640×2688×640	92K (10%)	806 (83%)	240 (52%)	400 (100%)	300	76.08	77.8	0.977	81.0%	123.9
2×6×2 (P2)	320×384×320	640×2304×640	91K (10%)	806 (83%)	240 (52%)	400 (100%)	300	76.08	77.5	0.982	73.2%	126.1
4×2×4 (P2)	**320×384×320**	**1280×768×1280**	**100K (11%)**	**912 (94%)**	**400 (86%)**	**400 (100%)**	**275**	**75.40**	**82.8**	**0.911**	**90.2%**	**100.6**
4×2×3 (P2)	320×384×320	1280×768×960	100K (11%)	912 (94%)	400 (86%)	400 (100%)	275	75.40	82.0	0.919	70.2%	109.7

TABLE IV: Evaluation of 10 top-ranked GEMM designs on Stratix 10 NX.

TB config.	Compute GEMM size	Native Buffer size	ALMs	BRAMs	TBs	Freq. (MHz)	Thrpt. (TOPs)	Power (W)	En. Eff. (TOPs/W)	RAM Eff.	BW (GB/s)
18×16×4×3	9×2720×4	639×2720×1008	124K (18%)	6304 (92%)	3456 (87%)	349	68.00	51.1	1.331	88.0%	92.6
18×8×8×3	9×1360×8	675×2720×928	123K (17%)	6064 (89%)	3456 (87%)	345	67.21	50.2	1.340	87.7%	91.6
9×16×5×5	**15×1280×5**	**900×1280×1000**	**127K (18%)**	**5840 (85%)**	**3600 (91%)**	**350**	**66.94**	**52.5**	**1.275**	**81.2%**	**90.2**
12×8×6×6	18×880×6	1152×1760×756	100K (14%)	6144 (90%)	3456 (87%)	338	64.00	48.6	1.317	86.7%	82.2
18×16×3×4	12×2720×3	850×2720×750	108K (15%)	6272 (92%)	3456 (87%)	327	63.71	47.3	1.347	85.9%	85.4
9×16×6×4	12×1280×6	912×2560×756	131K (19%)	6464 (94%)	3456 (87%)	342	62.88	50.7	1.241	85.1%	82.3
18×8×3×8	24×1360×3	1600×1360×550	81K (12%)	6064 (89%)	3456 (87%)	321	62.40	46.5	1.342	83.1%	92.4
9×8×10×5	**15×640×10**	**900×1280×1000**	**124K (18%)**	**5840 (85%)**	**3600 (91%)**	**320**	**61.21**	**48.7**	**1.257**	**81.2%**	**82.4**
18×8×5×5	15×1360×5	1020×2720×630	101K (14%)	6150 (90%)	3600 (91%)	301	61.08	45.4	1.346	90.0%	83.5
18×4×8×6	18×680×8	1152×1360×832	91K (13%)	6080 (89%)	3456 (87%)	312	60.69	46.2	1.315	84.3%	79.3

frequency and require a longer time for PnR, as they have less flexibility during placement. Note that design 9×16×5×5 has a lower throughput than 18×16×4×3, despite having higher TB utilization and operating at a higher frequency. This is attributed to a smaller TB_{len}, and therefore more wasted TBs operating in parallel load mode. Notice also how designs 18×16×4×3 and 18×8×8×3 perform better than designs 18×16×3×4 and 18×8×3×8, respectively. They achieve a higher frequency, due to a lower M_p, which leads to lower overall signal fan-out and FPGA routing congestion.

C. Insights & Discussion

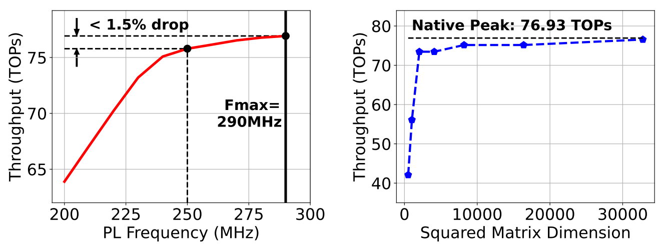
1) FPGA Frequency: We examine the reliance of our solutions on the attainable FPGA frequency. While Versal's AIE array operates at a fixed frequency (1.25 GHz), the PL (FPGA) frequency varies from 275-300 MHz across all top solutions (Table III). Despite the PL frequency variation, we observe a stable performance (∼2%) for Versal. We further depict the impact of PL frequency on throughput in Fig. 7a for design 2×2×8 ($P1$). We notice a negligible throughput decrease (<1.5%) for a wide frequency range (290 to 250 MHz). This is ascribed to higher AIE computation time compared to communication with PL, illustrating the *weak* dependence of performance on a *wide* PL frequency range. However, for frequencies lower than 250 MHz performance degrades more severely (∼16% from 250 to 200 MHz).

Conversely, for Stratix, we observed a higher performance range (∼12%) across all solutions in Table IV. This is attributed to the direct relationship of performance to frequency, illustrating the *strong* dependence of NX on PL frequency.

2) GEMM Scalability: Furthermore, we explore the scalability of our solutions when altering the matrix dimensions. For Versal, in Fig. 7b, we present the throughput of the best overall design 2×2×8 ($P1$), for square matrices (powers-of-2) ranging from 512 to 32K. Zero-padding is applied to align to the *compute* GEMM size (Table III). We notice that our design scales effectively, leading to almost its *native* peak throughput for dimensions ∼2K and higher. We note here that *native* peak refers to the *achieved* throughput of our designs when running GEMM on their *native* buffer sizes (Tables III, IV).

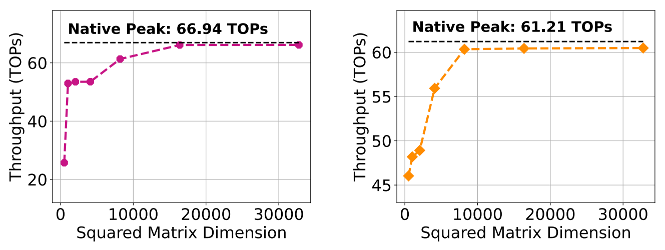
Similarly, for Stratix, we show the scalability of the two bolded designs in Table IV. The first (Fig. 8a) has a high $D_{K'}$, and presents one of the highest throughput among all top solutions. The second (Fig. 8b) has the lowest $D_{K'}$. As $D_{K'}$ is much higher than $D_{M'}$ and $D_{N'}$ (Table IV), designs with flexible (small) $D_{K'}$ require less zero-padding along the reduction dimension (K') and scale better. We observe that, while the design in Fig. 8a has higher *native* peak throughput, the design in Fig. 8b scales better, due to lower $D_{K'}$. Overall, GEMM scalability on both devices directly depends on the *compute* GEMM size. By properly adjusting this size, our methods can be exploited in straightforward fashion to target specific matrix dimensions *e.g.*, long and narrow matrices. However, for the sake of generality, we demonstrate GEMM scalability on square matrices, without targeting specific sizes.

3) Achieved Performance: As mentioned, Versal achieves ∼60% of its *theoretical peak* throughput. This can be ascribed to multiple reasons. First, the non-ideal (95%) efficiency [27], [28] of the AIE MatMul kernels. Second, the necessity to introduce Add kernels on the AIE, which do not contribute to throughput, but occupy some of the cores (Sec. IV-A2). Third, the inevitable stalls caused by memory conflicts on the AIE array [52]. In contrast, Stratix achieves ∼47% of its *theoretical peak* throughput. This is because the *theoretical peak* assumes 100% TB utilization with no wasted TBs in cascade mode, and



(a) (b)

Fig. 7: Performance *vs.* PL frequency (a), and square matrix dimensions (b) of 2×2×8 ($P1$) on Versal VC1902.



(a) (b)

Fig. 8: Performance *vs.* square matrix dimensions of designs 9×16×5×5 (a) and 9×8×10×5 (b) on Stratix 10 NX.

a very high operating frequency of 600MHz [10], which are infeasible to attain in practice.

Our results in Tables III, IV indicate that, on average, Versal attains 19.8% higher throughput, while Stratix presents 41.6% higher energy efficiency on GEMM workloads. However, we note that the focus of this study strays from a competitive comparison among the two AI-optimized FPGA architectures. Instead, we emphasize the distinct design methods required for each device, due to their considerably different architecture styles (out-of-fabric *vs.* in-fabric). A broader set of full AI-workloads would be needed to explore complicated trade-offs between the two architectures, which we leave as future work.

4) Programmability Trade-Offs: The two technologies require completely different programming methods. In particular, Versal utilizes high-level C/C++ software for programming the AIE array. Efficient, high-level programming constructs can enable high performance of the AIE array's SIMD processors on vectorizable workloads (with 95% AIE kernel efficiency). For productivity purposes, the PL part is typically programmed in HLS. However, HLS inefficiencies can lead to performance degradation, which can be mitigated by exploiting sophisticated methods (Sec. IV-A4). Finally, AMD/Xilinx provides automation tools, such as Vitis V++, to reduce the complexity of integrating the entire system.

On the contrary, no high-level tools are available for programming Stratix 10 NX. In particular, Intel HLS [56] has no support for TBs, thus requiring exclusive coding in RTL for design, verification, and full-system integration. This resulted in ∼40× higher number of lines of code in Stratix compared to Versal. Nonetheless, our Python-based generator significantly reduced this complexity. Moreover, because the performance of Stratix is directly analogous to frequency, unlike Versal, more programming effort is required to meet timing goals, and an extensive DSE is needed to identify high-throughput designs. In our case, this led to ∼5× higher total tool compilation time for Stratix compared to Versal. While each individual design required 3–6 hours to compile on both devices, the extensive exploration on Stratix greatly increased the design space. Finally, based on our estimations, it might be 1.5–2× more productive to program Versal due to the increased programming effort required by Stratix. However, we note that design productivity is highly dependent on designer skills.

5) GEMM Optimization Insights: Both devices exhibit design challenges due to their high complexity. To address complexity, systematic methodologies, *e.g.*, analytical modeling, are essential for optimization of GEMM-based applications.

Versal AIE is a novel architecture that introduces several new design challenges. A comprehensive exploitation of AIE architectural attributes, *e.g.*, local memory sharing and circuit switching, is crucial to optimize performance in GEMM [27]. Moreover, challenges, such as the reduced number of AIE-PL tiles and AIE routing congestion, require refined techniques to prevent performance degradation (Sec. IV-A2). Finally, the limited DRAM BW of Versal devices introduces additional challenges in maintaining the high throughput of the AIEs.

In contrast, Stratix 10 NX preserves a traditional FPGA architecture, while introducing TBs embedded in the PL. The inherent complexity of TBs necessitates the consideration of multiple operating modes to effectively map GEMM workloads (Sec. IV-B). Moreover, as throughput is directly related to frequency, performance depends on both low-level RTL optimizations and the intricacies of the FPGA architecture. RTL techniques, *e.g.*, replication of control logic and insertion of pipeline stages, are essential in shortening critical paths.

VI. Conclusion

In this work, we propose novel methodologies/frameworks to optimize GEMM-based applications targeting the two leading AI-optimized FPGA architectures. We present a thorough evaluation of several aspects in GEMM design, by efficiently leveraging the unique and substantially different architectural attributes of the AMD/Xilinx Versal ACAP and Intel Stratix 10 NX devices. Our experimental results show that our frameworks achieve up to 77 and 68 TOPs throughput on GEMM workloads for Versal and Stratix, respectively. This study provides fundamental insights regarding the architectural characteristics, programmability trade-offs, challenges and limitations inherent in both Versal and Stratix AI-optimized FPGAs.

Acknowledgment

We thank the anonymous reviewers for their insightful feedback, which improved the quality of the paper. This work was supported in part by the National Science Foundation CCF Grant No. 2107085, ExxonMobil Technology and Engineering Company, agreement no. EM10480.36, iMAGiNE - the Intelligent Machine Engineering Consortium at UT Austin, and a UT Cockrell School of Engineering Doctoral Fellowship.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, and et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[2] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Xie, B. Chen, C. Barrett, J. E. Gonzalez, P. Liang, C. Ré, I. Stoica, and C. Zhang, "FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU," 2023.

[3] "NVIDIA A100 Tensor Core GPU Architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[4] "Confidential Compute on NVIDIA Hopper H100," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf, 2023.

[5] "AMD CDNA 3 Architecture," https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf, 2023.

[6] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1–14.

[7] A. Firoozshahian, J. Coburn, R. Levenstein, R. Nattoji, A. Kamath, O. Wu, G. Grewal, H. Aepala, B. Jakka, B. Dreyer, A. Hutchin, U. Diril, K. Nair, E. K. Aredestani, M. Schatz, Y. Hao, R. Komuravelli, K. Ho, S. Abu Asal, and et al., "MTIA: First Generation Silicon Targeting Meta's Recommendation Systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589348

[8] "Goya Inference Platform White Paper," https://habana.ai/wp-content/uploads/pdf/habana_labs_goya_whitepaper.pdf, 2019.

[9] "Introducing Amazon EC2 Inf2 Instances Featuring AWS Inferentia2," https://d1.awsstatic.com/events/Summits/reinvent2022/CMP334_22986.pdf, 2022.

[10] M. Langhammer, E. Nurvitadhi, B. Pasca, and S. Gribok, "Stratix 10 NX architecture and applications," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 57–67.

[11] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx Adaptive Compute Acceleration Platform: Versal Architecture," ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 84–93. [Online]. Available: https://doi.org/10.1145/3289602.3293906

[12] E. Nurvitadhi, J. Cook, A. Mishra, D. Marr, K. Nealis, P. Colangelo, A. Ling, D. Capalija, U. Aydonat, A. Dasu, and S. Shumarayev, "In-Package Domain-Specific ASICs for Intel Stratix 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 106–1064.

[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, "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)*, 2019, pp. 199–207.

[14] A. Cairncross, B. Henry, C. Chalmers, D. Reid, J. Shipton, J. Fowler, L. Corrigan, and M. Ashby, "AI Benchmarking on Achronix Speedster® 7t FPGAs," 2023.

[15] A. Arora, S. Mehta, V. Betz, and L. K. John, "Tensor Slices to the Rescue: Supercharging ML Acceleration on FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 23–33. [Online]. Available: https://doi.org/10.1145/3431920.3439282

[16] A. Arora, M. Ghosh, S. Mehta, V. Betz, and L. K. John, "Tensor Slices: FPGA Building Blocks For The Deep Learning Era," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, aug 2022. [Online]. Available: https://doi.org/10.1145/3529650

[17] A. Arora, A. Bhamburkar, A. Borda, T. Anand, R. Sehgal, B. Hanindhito, P.-E. Gaillardon, J. Kulkarni, and L. K. John, "CoMeFa: Deploying Compute-in-Memory on FPGAs for Deep Learning Acceleration," *ACM Transactions on Reconfigurable Technology and Systems*, 2023.

[18] 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," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 94–103. [Online]. Available: https://doi.org/10.1145/3289602.3293912

[19] M. Eldafrawy, A. Boutros, S. Yazdanshenas, and V. Betz, "FPGA Logic Block Architectures for Efficient Deep Learning Inference," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 3, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3393668

[20] 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.

[21] S. Ahmad, S. Subramanian, V. Boppana, S. Lakka, F.-H. Ho, T. Knopp, J. Noguera, G. Singh, and R. Wittig, "Xilinx First 7nm Device: Versal AI Core (VC1902)," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–28.

[22] G. Alok, "Architecture apocalypse dream architecture for deep learning inference and compute-versal ai core," in *Embedded World Conference*, 2020.

[23] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, "Fathom: Reference workloads for modern deep learning methods," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[24] Y. E. Wang, C.-J. Wu, X. Wang, K. Hazelwood, and D. Brooks, "Exploiting Parallelism Opportunities with Deep Learning Frameworks," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, Dec 2021. [Online]. Available: https://doi.org/10.1145/3431388

[25] "VCK5000 Data Center Acceleration Development Kit Hardware Installation Guide (UG1531)," https://docs.xilinx.com/r/en-US/ug1531-vck5000-install/Card-Features.

[26] M. Adhiwiyogo, R. D'Souza, S. Leibson, and R. Shah, "Pushing AI boundaries with scalable compute-focused FPGAs," *Intel, White Paper*, 2019.

[27] E. Taka, A. Arora, K. C. Wu, and D. Marculescu, "MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine," in *2023 International Conference on Field-Programmable Technology (ICFPT)*, 2023, pp. 95–104.

[28] J. Zhuang, J. Lau, H. Ye, Z. Yang, Y. Du, J. Lo, K. Denolf, S. Neuendorffer, A. Jones, J. Hu, D. Chen, J. Cong, and P. Zhou, "CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 153–164. [Online]. Available: https://doi.org/10.1145/3543622.3573210

[29] J. Zhuang, Z. Yang, and P. Zhou, "High Performance, Low Power Matrix Multiply Design on ACAP: from Architecture, Design Challenges and DSE Perspectives," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[30] X. Jia, Y. Zhang, G. Liu, X. Yang, T. Zhang, J. Zheng, D. Xu, H. Wang, R. Zheng, S. Pareek, L. Tian, D. Xie, H. Luo, and Y. Shan, "XVDPU: A High Performance CNN Accelerator on the Versal Platform Powered by the AI Engine," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 01–09.

[31] T. Zhang, D. Li, H. Wang, Y. Li, X. Ma, W. Luo, Y. Wang, Y. Huang, Y. Li, Y. Zhang, X. Yang, X. Jia, Q. Lin, L. Tian, F. Jiang, D. Xie, H. Luo, and Y. Shan, "A-U3D: A Unified 2D/3D CNN Accelerator on the Versal Platform for Disparity Estimation," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 123–129.

[32] C. Zhang, T. Geng, A. Guo, J. Tian, M. Herbordt, A. Li, and D. Tao, "H-GCN: A graph convolutional network accelerator on versal ACAP architecture," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 200–208.

[33] P. Chen, P. Manjunath, S. Wijeratne, B. Zhang, and V. Prasanna, "Exploiting On-chip Heterogeneity of Versal Architecture for GNN Inference Acceleration," 2023.

[34] P. Chatarasi, S. Neuendorffer, S. Bayliss, K. Vissers, and V. Sarkar, "Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–10.

[35] Z. Yang, J. Zhuang, J. Yin, C. Yu, A. K. Jones, and P. Zhou, "AIM: Accelerating Arbitrary-precision Integer Multiplication on Heterogeneous Reconfigurable Computing Platform Versal ACAP," 2023.

[36] G. Singh, A. Khodamoradi, K. Denolf, J. Lo, J. Gomez-Luna, J. Melber, A. Bisca, H. Corporaal, and O. Mutlu, "SPARTA: Spatial Acceleration for Efficient and Scalable Horizontal Diffusion Weather Stencil Computation," in *Proceedings of the 37th International Conference on Supercomputing*, ser. ICS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 463–476. [Online]. Available: https://doi.org/10.1145/3577193.3593719

[37] N. Brown, "Exploring the Versal AI Engines for Accelerating Stencil-Based Atmospheric Advection Simulation," ser. FPGA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 91–97. [Online]. Available: https://doi.org/10.1145/3543622.3573047

[38] A. Boutros, E. Nurvitadhi, R. Ma, S. Gribok, Z. Zhao, J. C. Hoe, V. Betz, and M. Langhammer, "Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 10–19.

[39] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14.

[40] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 629–634.

[41] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn, and M. Blott, "FINN-L: Library Extensions and Design Trade-Off Analysis for Variable Precision LSTM Networks on FPGAs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 89–897.

[42] Z. Que, H. Nakahara, E. Nurvitadhi, H. Fan, C. Zeng, J. Meng, X. Niu, and W. Luk, "Optimizing Reconfigurable Recurrent Neural Networks," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 10–18.

[43] 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, "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)*, 2019, pp. 199–207.

[44] M. Stan, M. Hall, M. Ibrahim, and V. Betz, "HPIPE NX: Boosting cnn inference acceleration performance with ai-optimized FPGAs," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, 2022, pp. 1–9.

[45] M. Hall and V. Betz, "From TensorFlow Graphs to LUTs and Wires: Automated Sparse and Physically Aware CNN Hardware Generation," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 56–65.

[46] M. Langhammer, S. Finn, S. Gribok, and B. Pasca, "Dense FPGA Compute Using Signed Byte Tuples," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 130–138.

[47] J. Shipton, J. Fowler, C. Chalmers, S. Davis, S. Gooch, and G. Coccia, "Implementing WaveNet Using Intel® Stratix® 10 NX FPGA for Real-Time Speech Synthesis," https://www.intel.de/content/dam/www/central-libraries/us/en/documents/wp-01304-implementing-wavenet-using-intel-stratix10-nx-fpga-for-real-time-speech-synthesis.pdf, 2021.

[48] "Versal ACAP Design Guide (UG1273)," https://docs.xilinx.com/r/2022.1-English/ug1273-versal-acap-design, 2022.

[49] "Versal ACAP AI Engine Architecture Manual (AM009)," https://docs.xilinx.com/r/en-US/am009-versal-ai-engine, 2023.

[50] "AI Engine API User Guide," https://www.xilinx.com/htmldocs/xilinx2022_1/aiengine_api/aie_api/doc/index.html, 2022.

[51] "AI Engine Kernel and Graph Programming Guide (UG1079)," https://docs.xilinx.com/r/2022.2-English/ug1079-ai-engine-kernel-coding/Overview?tocId=OerrcATBJkz9SuXKjosb1w, 2022.

[52] "Versal ACAP AI Engine Programming Environment User Guide (UG1076)," https://docs.xilinx.com/r/2022.1-English/ug1076-ai-engine-environment/Overview, 2022.

[53] "Vitis High-Level Synthesis User Guide (UG1399)," https://docs.xilinx.com/r/2022.1-English/ug1399-vitis-hls, 2022.

[54] "Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)," https://docs.xilinx.com/r/2022.1-English/ug1393-vitis-application-acceleration, 2022.

[55] "Intel Stratix 10 Embedded Memory User Guide," https://www.intel.com/content/www/us/en/docs/programmable/683423/23-2/embedded-memory-configurations.html, 2023.

[56] "Intel High Level Synthesis Compiler Pro Edition: Reference Manual," https://www.intel.com/content/www/us/en/docs/programmable/683349/23-4/pro-edition-reference-manual.html, 2023.

[57] "MaxEVA," https://github.com/enyac-group/MaxEVA, 2023.

[58] "Versal AI Core Series Data Sheet: DC and AC Switching Characteristics (DS957)," https://docs.xilinx.com/r/en-US/ds957-versal-ai-core/AI-Engine-Switching-Characteristics, 2023.

[59] "Versal ACAP Memory Resources Architecture Manual (AM007)," https://docs.xilinx.com/r/en-US/am007-versal-memory, 2020.

[60] N. Voss, P. Quintana, O. Mencer, W. Luk, and G. Gaydadjiev, "Memory Mapping for Multi-die FPGAs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 78–86.

[61] D. Karchmer and J. Rose, "Definition And Solution Of The Memory Packing Problem For Field-programmable Systems," in *IEEE/ACM International Conference on Computer-Aided Design*, 1994, pp. 20–26.

[62] R. Tessier, V. Betz, D. Neto, A. Egier, and T. Gopalsamy, "Power-Efficient RAM Mapping Algorithms for FPGA Embedded Memory Blocks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 278–290, 2007.

[63] J. Vasiljevic and P. Chow, "Using buffer-to-BRAM mapping approaches to trade-off throughput vs. memory use," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[64] "Vivado Design Suite User Guide: Power Analysis and Optimization (UG907)," https://docs.xilinx.com/v/u/2019.1-English/ug907-vivado-power-analysis-optimization, 2022.

[65] "ModelSim User's Manual," https://www.microsemi.com/document-portal/doc_view/131619-modelsim-user, 2023.

[66] "Intel Quartus Prime Standard Edition User Guide: Power Analysis and Optimization," https://www.intel.com/content/www/us/en/docs/programmable/683506/18-1/power-analysis.html, 2018.