Performance Analysis of GEMM Workloads on the AMD Versal Platform

Kaustubh Manohar Mhatre
Arizona State University
Tempe, USA
kmhatre@asu.edu

Venkata Guru Prashanth Mulleti
Arizona State University
Tempe, USA
vmulleti@asu.edu

Curt John Bansil
Arizona State University
Tempe, USA
vmulleti@asu.edu

Endri Taka
The University of Texas at Austin
Austin, USA
endri.taka@utexas.edu

Aman Arora
Arizona State University
Tempe, USA
aman.kbm@asu.edu

***Abstract*—AMD Versal is a new heterogeneous computing hardware architecture comprised of adaptive intelligence (AI) engines, programmable logic, and a processing system. General Matrix Multiplication (GEMM) is the fundamental building block of modern deep learning (DL) applications such as ChatGPT, and GEMM workloads can be mapped onto Versal in different ways, each with distinct trade-offs. This paper presents a thorough analysis of GEMM workloads of different shapes and sizes, showcasing performance artifacts associated with the AMD Versal architecture. Focusing on the unique aspects of the Versal architecture, multiple research questions related to performance scaling, sensitivity, and efficiency are explored. This paper aims to assist FPGA developers looking to implement GEMM on AMD Versal by providing insights for enhancing performance.**

***Index Terms*—Versal, Heterogeneous Architecture, Hardware Accelerator, Matrix Multiply, Deep Learning**

I. Introduction

The progress of DL in the past decade has increased the demand for more energy-efficient computing. GEMM, at the heart of DL, has become much more vital in recent Transformer-based models that power mainstream applications such as ChatGPT compared to Convolutional Neural Networks. GEMM constitutes more than 90% of the compute operations in Transformers [15]. As such, optimizing GEMM can significantly improve the performance of DL workloads. Hardware acceleration of GEMM is commonplace - by using dedicated ASICs like Google TPU [11] or by adding blocks such as Tensor Cores in NVIDIA GPUs [1]. AMD introduced a new heterogeneous architecture called AMD Versal [10] that includes Programmable logic (PL), Processing system (PS), and AI engines (AIEs). AIEs are grouped into a 2D array of software-programmable vector processors that operate at a frequency much higher than PL, and provide a significantly higher compute throughput. AIEs are connected to the PL using special interfaces called PLIOs. The Versal architecture also has a network on-chip (NoC) that connects all of these elements to the DRAM. The memory architecture is multi-level, starting with AI engine's tightly coupled data memory, PL memory, and lastly the DRAM. The heterogeneity of the resources in AMD Versal makes programming it complex, especially when optimizing for performance.

GEMM significantly contributes to execution time on the Versal architecture for deep learning workloads, as shown by CHARM [19] and SSR [21]. In this work, we analyze various performance artifacts of mapping GEMM kernels onto this unique architecture. To the best of our knowledge, no prior work has delved into such a detailed analysis of GEMM workloads on Versal. We extensively analyze various sizes and shapes of GEMM workload running on AMD Versal with varying resources like AIEs and multi-level memory hierarchy. We focus on the unique aspects of the Versal architecture, such as PLIO usage, data transfer patterns between PL and AIEs, different implementations, and buffer utilization of AIEs.

Such performance analysis can be immensely useful in increasing the adoption of Versal architecture. There are numerous ways of mapping a GEMM workload to the Versal architecture with tradeoffs that are not easily and analytically evident. It is not easy to understand the contributions of various architectural features to performance. There is a lack of resources that detail which approach leads to higher performance, more scalable implementation, lower energy, etc. Our goal is to bridge this gap.

Our contributions in this paper are:

- We pose a set of research questions regarding the performance of GEMM workloads on AMD Versal, set up experiments to study them, and identify insights on hardware and software characteristics.
- We analyze the performance of GEMM kernels mapped to a single AI Engine by running various GEMM sizes and shapes, while providing a breakdown of execution time spent on communication and compute.
- We analyze the performance of GEMM workloads mapped to multiple AIEs. We demonstrate the impact of workload scaling with different configurations that utilize various resources (AI Engine, PL, PLIO).
- We offer an analytical model that provides performance estimates, execution breakdown, and insights into potential bottlenecks in the design.

II. Related work

Multiple prior works have deployed ML workloads on AMD Versal. **CHARM** [19] is a framework for accelerating Deep Neural Networks (DNNs) on Versal, using the AIE array and PL. It divides the AIE array into multiple accelerators based on matrix multiplication size and memory requirement, targeting only FP32 precision. **AutoMM** [22] builds on top of CHARM [19] with a Python based interface to streamline accelerator design and uses INT8 precision. **CHARM 2.0** [20] adds support for INT16 precision as well. (*CHARM, AutoMM and CHARM 2.0 are available in the same codebase, so we do not differentiate between them in this paper and refer to them as CHARM, to avoid confusion*). **MAXEVA** [14] improves over these works to provide higher performance. CHARM's methodology presents a resource-efficient approach to design, whereas MAXEVA achieves high performance but at the expense of significantly increased resource utilization, thereby reducing scalability and limiting its feasibility to only small designs. **SSR** [21] deploys DNNs with smaller memory footprint which can be stored on-chip on Versal. This enables them to implement optimizations like on-chip weight and output activation storage to reduce the DRAM pressure. **AIM** [17] focuses on arbitrary-precision integer multiplication, primarily associated with scientific computing, on Versal and shows that the combination of AIE and PL can increase energy efficiency compared to CPU and GPU. **H-GCN** [18] implements graph neural networks on Versal, performing sparse matrix multiplication (SpMM) on the AIE array. **Chen et al.** [9] demonstrate GNN implementation on Versal with performance gains compared to CPU and GPU. **Vyasa** [8] enhances the programmability by extending the Halide DSL compiler to automatically generate code for AIEs. **SPARTA** [13] implements weather prediction using AI Engines on AMD Versal leveraging the MLIR framework. Whereas [7] uses them for atmospheric simulations. **Perryman et al.** [12] demonstrate the use of AIE for space edge computing applications (CNNs) due to high energy efficiency compared to programmable logic.

These works do not do a thorough performance analysis of the Versal hardware. They focus on accelerating a specific workload without giving insights into bottlenecks. The work by **Wierse** [16] represents the closest comparable research to ours, as it specifically focuses on evaluating the communication pathways within the chip, from the DRAM to the AIE. However, it does not extend beyond the evaluation of these interconnects, whereas our work also focuses on compute efficiency and performance.

III. Overview of Versal Architecture

Figure 1 shows the Versal AIE architecture[3]. We use VCK5000 for our experiments and discuss the important components below.

Processing System: Versal features an ARM dual core Cortex A-72 processor, which serves as a host. This core is a programming interface for the AIEs and the PL. Both PS and PL can be controlled using the PS. It can run a Linux operating system, enabling it to run a wide range of operations.

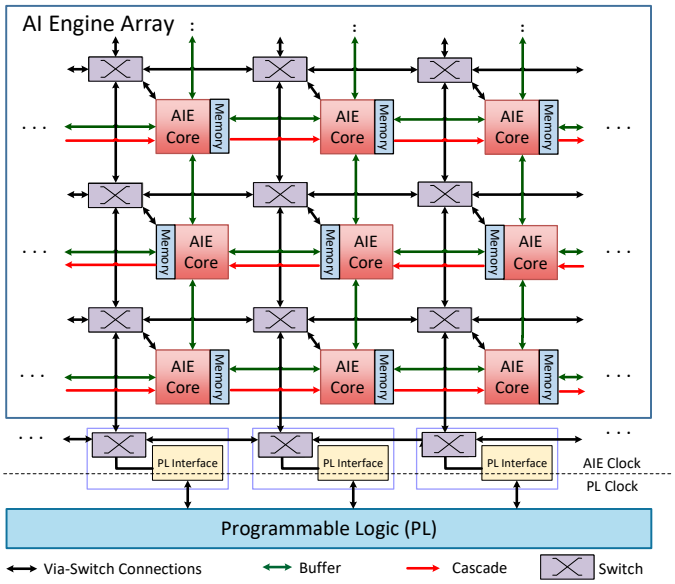

Fig. 1: Versal AIE architecture

Programmable Logic (PL): The PL comprises the FPGA reconfigurable fabric with Lookup Tables (LUTs), flip-flops (FFs), Digital Signal Processing Slices (DSPs), Block RAMs (BRAMs), and Ultra RAMs (URAMs). It is capable of implementing any custom datapath necessary for an application.

AIEs: The AIEs consist of an array of vector processors operating at a frequency of 1.25GHz. Each AIE includes 32KB of tightly coupled memory for storing program instructions and data. Figure 1 shows the architecture of the AIE array and various connectivity interfaces between AIEs (Via-Switch Connections, Buffer and Cascade). The vector processor within the AIE achieves 8 MACs per cycle for FP32 and 128 MACs per cycle for INT8 operations.

AIE Interfaces: AIEs have two main interfaces in the last row of the AIE array and are of two types: NoC interface blocks and PL interface blocks. Connections to DRAM are made from the NoC interface bocks, whereas connections to PL can be made from both blocks. The PL to AIE interface is called PLIO. The bit width of a PLIO interface is 64-bit, but can also be configured to 128-bit (at 0.5× the frequency).

Memory Hierarchy: The AMD Versal has 3 levels of memory hierarchy. Every AIE has its own internal memory of 32KB for program and data. There is a total of 12.8 MB of AIE internal memory if all 400 AIEs are used. The next level is the PL memory, which consists of BRAMs and URAMs. There are 967 BRAMs, each of size 36 Kbit, totaling up to 4.6 MB of Memory. There are 463 URAMs each of size 288Kbit totaling up to 17.1 MB. The final level is the DRAM. There is 16 GB of DDR4 memory on the board.

Programming Model[5]: AIE kernels are programmed in C/C++ either (1) using high-level APIs provided by AMD [2] that handle common operations like matrix multiplication and FIR, or (2) using intrinsics [4] which are low-level

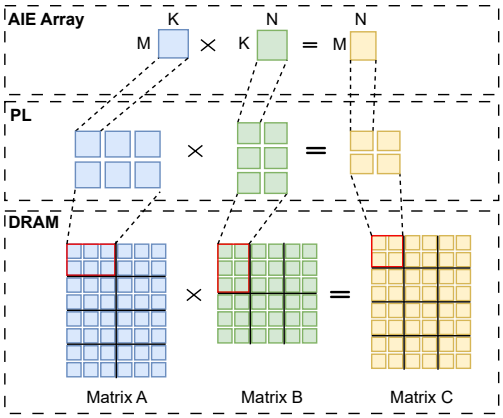

Fig. 2: GEMM tiling. The three tiling stages involves transferring matrices A and B from DRAM to PL, and subsequently to AIE.

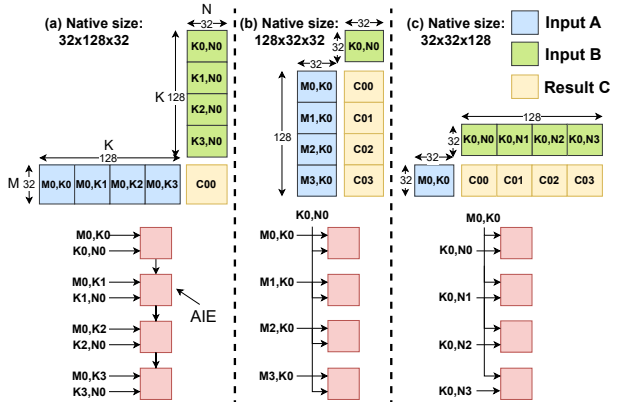

Fig. 3: Different AIE connectivity leads to different native size.

architecture-specific instructions that offer finer control.

Speeds and Feeds: VCK5000 has 400 AIEs running at 1.25 GHz, resulting in a theoretical throughput of:

$$FP32_{thrpt} = AIE_{freq} * PeakThrpt_{FP32} * \#AIEs * 2(*,+)$$
$$= 1.25GHz * 8 * 400 * 2 = \textbf{8 TFLOPs}$$

$$INT8_{thrpt} = AIE_{freq} * PeakThrpt_{INT8} * \#AIEs * 2(*,+)$$
$$= 1.25GHz * 128 * 400 * 2 = \textbf{128 TOPs}$$

Each PLIO interface, if running at 500 MHz, will support a BW of 4000 MB/s.

$$SinglePLIOBandwidth = 500MHz * 64bit = \textbf{4000 MB/s}$$

Every block has 8 connections from PL to AIE and 6 from AIE to PL. Thus, the total BW on the AIE-PL interface is

$$PLtoAIE = 4GB/s * 8 * 39 = \textbf{1.2 TB/s}$$

$$AIEtoPL = 4GB/s * 6 * 39 = \textbf{0.9 TB/s}$$

DRAM Bandwidth: VCK5000 is equipped with four 4GB DDR4 72-bit interfaces that can operate at 3200 Mb/s, providing a bandwidth of 102 GB/s. The access to the DDR is provided through the integrated Network On Chip (NoC) and can be done either from the PL or the AIEs.

IV. Methodology

A. Mapping GEMM on Versal

A workload can be mapped on AMD Versal in several ways considering resource utilization and performance requirements. We explain our implementation of GEMM here. As shown in Figure 2, the input matrices are present in the DRAM. They are read from the DRAM and buffered into the PL memory (BRAM and URAM) using DMA, and then moved to the internal memory of the AIEs using PLIO interfaces. The vector processor of the AIE reads from the AIE's memory to perform the computation. The kernel that runs inside the AIE performs matrix multiplication using a basic three-loop pattern; the innermost loop iterates over the reduction dimension (K), and the other two loops over the input rows (M) and output columns (N). The input read overlaps with the compute to provide high efficiency.

Running large GEMM workloads requires breaking down the workload into small parts, known as **tiling**, to take advantage of the data reuse inherent in the GEMM operation. The three levels of memory hierarchy (DRAM, PL memory and AIE memory) enable us to perform tiling at three distinct levels: DRAM, PL and AIE. Figure 2 depicts this multi-level tiling flow where chunks of matrix A and matrix B are copied to the PL. These chunks present in PL are further broken down and copied into the AIE memory to perform final computation. The size of these chunks is called **tile size**, which is influenced by the hardware configuration such as number of AIEs and PL memory. At the PL to AIE level, the tile size is governed by the group of AIE engines and its connectivity (4 engines connected using cascade connections), whereas at the DRAM to PL level, it is governed by the maximum size of PL memory and the number of AIE groups, while being a multiple of the PL to AIE tile size. Tiling any GEMM operation comes at the cost of reading the same data multiple times. This excess communication is called tiling overhead. Tiling overhead at the DRAM level is the costliest, which we aim to reduce as we have limited DRAM BW. Tiling at the PL level has a reduced impact on performance, as PL-AIE bandwidth is much higher. The AIE kernel performs tiling inside the kernel to run the computation. Tiling overhead at this level minimally impacts the performance much because of faster access. **Double buffering** is used to overlap communication between PL and AIEs and the compute. We do double buffering for both inputs and outputs to ensure complete overlap. This doubles the memory requirement in the AIE, and restricts the workload size that can be run inside a single AIE. Double buffering is also performed at the DRAM-PL interface inside PL to overlap communication. Multiple levels of tiling add to the complexity of reduction at the AIE level as well as PL level. For evaluation, we use synthetic workloads as well as workloads from popular DNNs. Our synthetic workload sizes are also influenced by the tile size (workload dimensions are integer multiples of the tile size), since our goal is to evaluate the highest compute throughput achievable in the

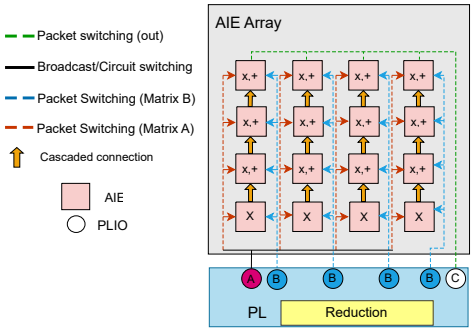

Fig. 4: CHARM connectivty diagram for 16 AIEs

Versal chip. This approach minimizes fragmentation/padding. The trade-offs between different tile sizes and their effects on fragmentation/padding for DNN workloads are left as future work.

Data transfer between PL and AIE is done using PLIO interfaces, which are a scarce resource. The simplest way of sending data is to connect a PLIO with one AIE (one-to-one connectivity). **Packet switching** dynamically transfers data from a PLIO to any AIE that is connected to the same PLIO. This is essentially achieved by having a header packet that stores the address of the AIE it wants to reach. This effectively time multiplexes the data transfer from one source (PLIO) to multiple sinks (AIEs). **Circuit switching** statically connects the same PLIO interface to multiple AIEs, thus supplying data directly to the engines. This is mainly used for broadcasting operations. Packet and circuit switching reduce the PLIO requirement compared to one-to-one communication. Packet switching trades off performance through serialization, while circuit switching enables deterministic latency and only applies when data is broadcasted.

Connections between the AIEs can be designed as per the application needs. Multiple AIEs can be grouped in various numbers and ways to perform GEMM operations. An increased number of AIEs enhances the capability of handling larger GEMM sizes. This grouping of the AIEs demands a minimum size of the workload that runs completely parallel on all the engines. We define this size as **the native size**. Figure 3 explains this with examples. On the top, three different GEMMs are shown with matrix dimensions, and each individual block represents a 32x32 chunk. At the bottom, the connectivity of AIEs is shown for those GEMMs. Figure 3 (a) illustrates the connectivity that leads to an expanded K dimension, thereby establishing a native size of 32x128x32[1]. Figures 3 (b) and (c), on the other hand, show connectivity that makes the dimensions of M and N longer, respectively. Workloads smaller than the native size are padded, while larger ones are divided into chunks of the native size.

CHARM [19] presents a complete framework for mapping and running GEMM workloads using its own design space exploration (DSE) tool that finds a balance between resource usage and performance. Mapping begins with every AIE having an individual kernel that runs a small matrix multiplication (32x32x32) for FP32 and (64x64x64) for INT8. These engines are then chained together into packs of 4 (FP32) and 2 (INT8) to perform matrix multiplication and reduction in a cascaded fashion. It uses a combination of circuit-switched and packet-switched connections to optimize the PLIO usage. In order to work efficiently, CHARM uses kernel sizes that overlap compute and communication for the pack of AIEs. The engine input is double buffered to overlap computation with PL to AIE level communication. CHARM uses one kernel performing dot product and reduction inside a cluster of AIEs. A reduction outside the cluster must be done in the PL. The cluster size for CHARM is fixed to 16 AIEs. The PL memory also helps reduce the DRAM pressure as it keeps partial results and reduces the tiling (or blocking) overhead. We use CHARM for our GEMM implementation.

[1]The notation MxKxN used throughout the paper denotes a matrix multiplication operation involving multiplying a MxK matrix with a KxN matrix resulting in an MxN matrix.

B. Research Questions for GEMM Implementation on Versal

GEMM workloads can be mapped to Versal's new reconfigurable architecture in multiple ways. However, irrespective of the implementation, there are research questions that arise about the performance aspects of the architecture. We use SOTA implementations of GEMM kernels for our analyses to analyze the performance of GEMM implementation with these research questions in mind:

- How much perf. can be achieved compared to the theoretical peak (i.e. what's the efficiency)? (Section V-C)
- How much is the overhead of data transfer (both DRAM to PL and PL to AIE) compared to compute? (V-G)
- How does the performance vary by changing the programming model (using intrinsics vs. API)? (V-B)
- How does the performance scale (weak scaling and strong scaling)? (V-E and V-F)
- How sensitive is performance to workload parameters (size, shape)? (V-C, V-E, V-F). How much performance can be achieved on tall/skinny matrix sizes that are common in real-world DNNs? (V-I)
- How sensitive is perf. to arch. parameters (# AIEs, # PLIOs, PL memory)? (V-H, V-E, V-F)
- What performance impact do different communication schemes between AIEs have? (V-D) (V-H)
- What are the maximum bounds on compute and memory on the Versal hardware? Are real-world workloads compute-bound or memory-bound? (V-J).

TABLE I: Versal execution platforms
(FV = Functional Verification, P = Performance)

Platform	Simulation Target	Speed	Usecase
aiesimulator	AIE + AIE ⇔ PL	Fast	FV+P
sw_emu	PL + AIE + Host	Fast	FV
hw_emu	PL + AIE + Host	Slow	FV+P
HW	PL + AIE + Host	Fast	FV+P
Analytical model	PL + AIE + Host	Fast	P

C. Experimental Setup

We use AMD Vitis 2022.2 for simulation, synthesis and implementation on VCK5000 using standard Vitis flow. In

our GEMM implementation, the AIEs perform the entire computation as described in Section IV-A. AIEs receive inputs from the PL and send outputs to the PL. The PL interfaces with the DRAM. The PL logic (i.e. interfacing with AIE and DRAM) is programmed using High-Level Synthesis (HLS). The AIE array runs at a frequency of 1.25 GHz, and the PL design operates at 230 MHz. The bitwidth of each PLIO interface is configured to be 128 bits and the bitwidth of the design's ports for performing reads and writes is 512 bits.

Platforms: There are four execution platforms [6] provided by AMD as listed in Table I. The **`aiesimulator`** is used for AIE graph simulation only. Results from the `aiesimulator` are cycle accurate and provide insights into the detailed breakdown of execution time within the AIEs and the data transfer between AIEs and PL. The **sw_emu** enables a complete application simulation that includes AIEs, PL and PS. This platform is only used for functional verification and has the least compile/debug time. The **hw_emu** does hardware emulation of the complete application. This execution platform is mainly used to get performance insights into the entire application. Its emulation speed is very slow. Lastly, the **HW** platform synthesizes the PL and packages the entire application to run on VCK5000. Due to long synthesis and implementation times, we limit our hardware runs to record only the final end-to-end performance.

DRAM interfacing: VCK5000 features just four vertical lanes for linking the PL with the NoC ports, each offering a bandwidth of 16 GB/s. Thus, if we read DRAM through all the lanes at the same time we can get 64 GB/s of bandwidth (same applies to writes). Each vertical lane consists of 8 interleaved virtual channels. Consequently, if after placement, all the design's ports (generated by HLS) connect to the same vertical lane, the bandwidth gets limited to 16 GB/s. During our analysis, we did not find a direct way to assign NoC ports to the design's ports through the Vitis design flow. The NoC compiler infers the connectivity based on the bandwidth requirements specified in the Quality of Service(QoS) settings of the NoC. However, specifying the right bandwidth does not assign the design ports to separate physical lanes.

We attempted to increase the utilization of the DRAM bandwidth by adding more ports on our design (through HLS pragmas). The existing CHARM configuration features two read ports and one write port (2r1w), resulting in 20 GB/s bandwidth utilization. Increasing this to four read ports and two write ports (4r2w) results in a bandwidth of 34 GB/s. Increasing the design's ports further did not enhance the DRAM bandwidth utilization. Thus, we could only achieve 34% bandwidth utilization on the chip. This limited bandwidth is primarily due to the design's ports being assigned to virtual channels of the same vertical NoC lane. This connectivity is not configurable through the Vitis design flow.

V. Results

A. Analytical model accuracy

We develop an analytical model by extending the CHARM's analytical model to generate performance estimates without going through the tedious synthesis and implementation process and to get insights into the execution breakdown. It first calculates the time consumed to complete the execution on the data present in the PL as given by Equation 1. The data transfer between the PL and AIE (both read and write) can be overlapped with AIE compute time due to double buffering. Hence, the max of the data transfers and the AIE compute time is taken and multiplied with the total tiles in the PL.

$$AIE\_CYCLES = \#PL\_Tiles * max(PL\_to\_AIE_A, PL\_to\_AIE_B, Time_{Compute}, AIE\_to\_PL_C) \quad (1)$$

$$Final\_Time = \#DRAM\_Tiles * max(DRAM\_to\_PL_A, DRAM\_to\_PL_B, AIE\_CYCLES, PL\_to\_DRAM_C) \quad (2)$$

Double buffering is also used in the PL. This helps to overlap the DRAM read and write with the AIE_CYCLES from Equation 1. The max of DRAM read, write and AIE_CYCLES is taken, and multiplied with the total number of tiles in DRAM to get the final time as given by Equation 2.

In our analytical model, we expand upon CHARM's framework to accommodate the bandwidth specifications of VCK5000. We improve the existing design by incorporating a greater number of parallel DRAM access ports and introducing "access ports" as an additional parameter for design space exploration. Our model also extracts execution breakdown, given a workload size and hardware configuration. This helps understand performance bottlenecks. To ensure the reliability of the model, we run several workloads on the hardware and compare the performance with the results from the analytical model. We calibrate the model to add a fixed setup duration of 100us that is consumed by the AIE engines. This increases the accuracy of estimation, especially for smaller kernels where the overall execution time is low. Our experiments show that the analytical model's estimate are within ±5% of the actual hardware execution time.

B. Comparison of kernel performance with intrinsics and API

We analyze the performance of the GEMM operation mapped to a single AIE. Kernel sizes of 32x32x32 (for FP32) and 64x64x64 (for INT8) are executed. These sizes provide the best performance for single AIE execution because they maximize the utilization of the AIE internal memory via double buffering. Furthermore, they provide significant overlap between communication and computation. More details on kernel size selection can be found in Section V-C. Figure 5 (left) shows the kernel efficiency for two different design methods, namely using API[2] and intrinsic[4]. Results here are obtained using `aiesimulator`. The API we use is called `aie::mmul` and the intrinsic we use is `mac16` (for int8) and `fpmac` (for FP32). Kernel efficiency is defined as the ratio of the theoretical time based on the peak throughput of the AIE to the observed execution time. Efficiencies over 90% are observed for both FP32 and INT8 precisions. For INT8, the efficiency of the API-based and intrinsic-based kernels

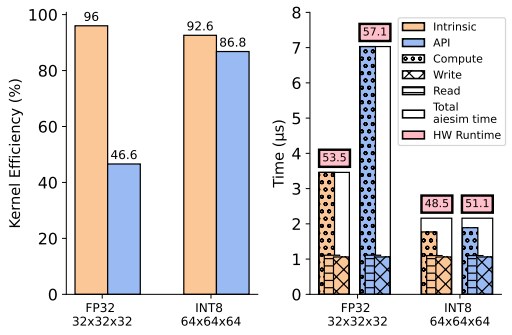

Fig. 5: Single AIE kernel results comparing implementations using intrinsic and API. FP32 APIs are not as efficient as INT8 APIs are, compared to intrinsics. Kernels with APIs show 46% and 7% reduction in performance for FP32 and INT8 respectively. Results obtained using `aiesimulator` and hardware runs.

do not differ significantly, but for FP32, the intrinsic-based kernel shows over 2× higher efficiency compared to the API-based kernel. Although lower in efficiency, API-based kernels offer high portability across various Versal devices (hardware-independent code) and simplify programming.

Figure 5 (right) shows a breakdown of the time consumed by the kernel. The read and write operations are stacked because of their sequential execution, while the compute operations are executed in parallel with the communication (read/write), as demonstrated by the overlapping bars. **Read** indicates data transfer from PL to AIE, and **Write** indicates data transfer from AIE to PL. Our intrinsic-based kernels clearly show significant overlap for both FP32 and INT8, whereas API-based kernel is heavily compute-bound for FP32, indicating further scope for improvement. The figure also shows the hardware execution time on top of each bar in a pink box. This time includes the time consumed by the HLS kernel execution in the PL and DRAM transfers, including the AIE execution and PL-AIE data transfer. The hardware execution time is higher than the `aiesimulator` time because of two reasons. Firstly, transferring data from DRAM to the PL consumes time, and the efficiency of DRAM bandwidth is low for smaller sizes, leading to a longer transfer time. Secondly, the AIE-to-PL communication has a non-overlapping data transfer overhead that cannot be overlapped with AIE compute.

Summary on kernel programming style (Intrinsic vs API): For best performance, intrinsics should be utilized as they offer finer control and low-level access; however, these kernels are device-specific and lack portability. When targeting multiple platforms, API-based kernels are recommended due to their ease of use and portability, with minimal performance loss for INT8. For FP32, using intrinsics is better until vendor's API implementation is improved. (Figure V-B)

C. Variation in single AIE perf. for different workload sizes

Figures 6 and 7 show the results of single AIE kernels for different shapes and sizes. We use symmetric (square matrices) and asymmetric (fat and skinny matrices) shapes to cover multiple performance scenarios. The total memory addressable by a single AIE amounts to 32KB × 4 = 128 KB, where

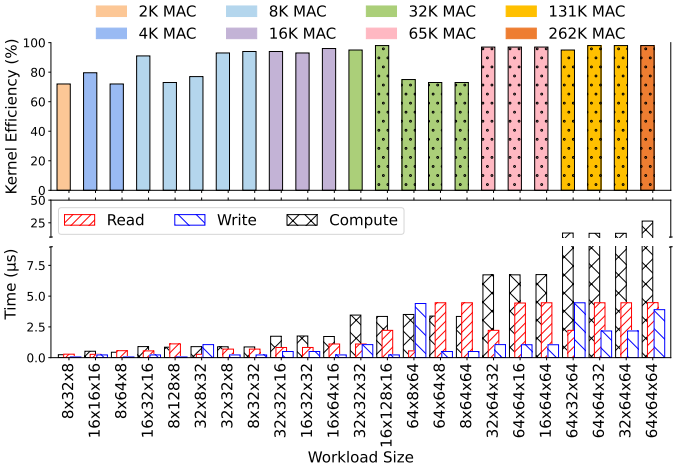

Fig. 6: Single AIE kernel efficiency (top) showing the effect of workload size and shape on efficiency. Bars with dots represent workload sizes that require memory from neighboring engines. The execution time breakdown (bottom) shows compute and communication overlap. Results obtained using `aiesimulator`. Precision=FP32.

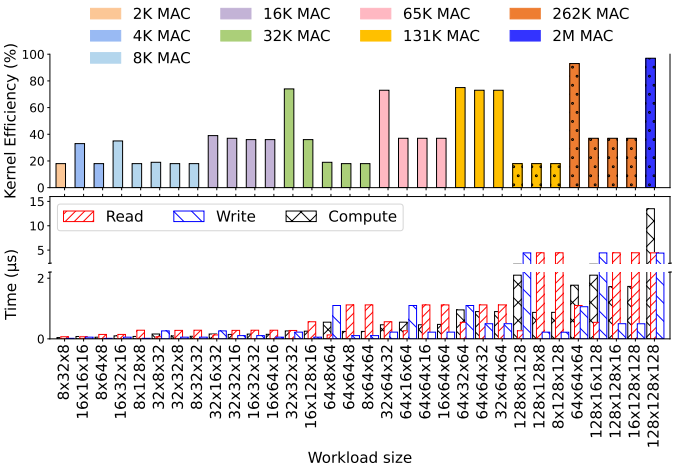

Fig. 7: Single AIE kernel efficiency (top) showing the effect of workload size and shape on efficiency. Bars with dots represent workload sizes that require memory from neighboring engines. The execution time breakdown (bottom) shows compute and communication overlap. Results obtained using `aiesimulator`. Precision=INT8.

32KB is its own memory and 96 KB are from neighboring AIEs. Using kernel sizes that incorporate neighboring memory enables one to employ larger kernels, marked by dots in Figure 6 and 7 (top). While larger kernels offer increased computational throughput, they are not scalable across the entire AIE array. When the kernels are small enough to fit within a single AIE, they can be easily scaled across the entire AIE array. To enhance performance, inputs are double-buffered to overlap computation with data transfer. However, the AIE execution model places a constraint on each individual double buffer to be contained within a single AIE. Therefore, we cannot have a double buffer occupying more than 32KB of space. This limits the individual matrix size to 16 KB, translating to 4k elements for FP32 or 16k for INT8. Thus, for a single AIE execution, the maximum workload size is 64x64x64 for FP32 and 128x128x128 for INT8. Disabling

the double buffering removes the overlap and makes the communication and compute operations sequential.

Kernel efficiency is defined as the ratio of the theoretical time based on the peak throughput of the AIE to the observed execution time (max of compute and communication). FP32 kernels (Figure 6 (top)) achieve efficiency from 70% to 98%. Workloads with higher communication time compared to compute time show low efficiency. For INT8 (Figure 7 (top)), only a few kernels show high efficiency. INT8 compute grows 16x (128 MACs_per_cycle / 8 MACs_per_cycle), while communication data reduces by only 4x (32 Byte / 8 Byte) compared to FP32, resulting in most kernels having higher communication time than compute.

Figure 6 (bottom) shows the execution and data transfer breakdown for various matrix shapes and sizes for FP32 precision. The communication from PL to AIE and AIE to PL overlaps with compute due to double buffering and is shown as an overlap in the graph. Our implementation uses separate PLIO resources for Matrix A, B, and Out. Thus, the read of Matrix A and B is overlapped, and the max of both reads is shown. Majority of the workloads are compute-bound as the FP32 throughput of AIEs is only 8 MAC/cycle. Figure 7 (bottom) shows the same breakdown for INT8 precision. Due to high INT8 throughput (128 MAC/cycle), most INT8 workloads are communication-bound except 128x128x128. For all further experiments, our kernels use 32x32x32 for FP32 and 64x64x64 for INT8, as these sizes provide high efficiency and excellent compute-communication overlap. These sizes fit within the local memory of a single AIE without requiring memory from neighboring AIEs, making them scalable across the AIE array.

Summary on selecting AIE kernel size: Kernel size and shape selection are highly design-specific. For optimal performance, kernels must be both efficient and scalable throughout the AIE array. For example, kernels such as 16x128x16 (for FP32) and 128x128x128 (for INT8) have the highest efficiency but require using the memory of neighboring AIEs. Scaling such kernels to the full AIE array can lead to underutilization of the array. Another design-specific factor is the overlap of memory (data transfer) with computation time. For example, in our case using 16x128x16 size for FP32 precision in a 4-AIE configuration does not allow for overlapping the compute and data transfer time (3.35us compute time vs. 8.8us data transfer time). Hence, choosing kernels with slightly lower efficiency, such as 64x64x64 for INT8 and 32x32x32 for FP32, can lead to better overall performance.

D. Effect of different communication schemes between AIEs

Communication between AIEs can be done using multiple interfaces such as Via-Switch Connections, Buffer, and Cascade connections (shown in Fig 1). Cascade connections directly communicate partial sums to the neighboring AIE. Via-Switch Connections transmit data using the switches from one AIE to any other AIE. Buffer connections enable accessing the memory of three neighboring engines directly. The usage of Buffer stalls the consumer AIE until the data is completely

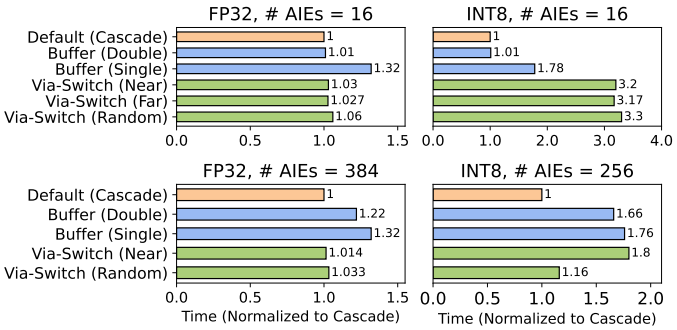

Fig. 8: Execution time comparison of different AIE to AIE communication schemes normalized to Cascade. Left is FP32 with 16 AIEs (Top) and 384 AIEs (Bottom) without the PL. Right is INT8 with 16 AIEs (Top) and 256 AIEs (Bottom) without the PL. Results obtained using `aiesimulator`

produced by the producer AIE. Buffers can be implemented either as a single Buffer (i.e., serializing read and write) or as a double Buffer (i.e., enabling simultaneous read and write). The double Buffer essentially overlaps the computation between two AIEs, enabling both to run simultaneously.

To quantify the difference in performance, we set up an experiment to perform matrix multiplication in the style shown in Figure 4. Figure 8 shows FP32 on the left and INT8 on the right. We first experimented with fewer AIEs (16 AIEs for both FP32 and INT8) and then with the maximum possible AIEs (384 AIEs for FP32 and 256 AIEs for INT8). For fewer AIEs, the double Buffer increases the execution time by 1% for both FP32 and INT8, while the single Buffer increases it by 32% and 78% for FP32 and INT8 respectively. When we use more flexible Via-Switch connections where the AIEs kernels are placed near, far, or random (i.e., placing of kernel is done by compiler) instead of near-neighbor cascade connections, we see upto 6% increase in execution time for FP32. For INT8, the Via-Switch connections show a 3.17-3.3x increase in execution time, significantly higher than FP32. This can be attributed to the fact that INT8 exhibits a compute throughput that is 16 times greater than that of FP32, thus making the execution time more sensitive to AIE to AIE communication.

When considering maximum possible AIEs for FP32, double Buffer increases the execution time by 22% while single Buffer increases the execution time by 32%. Via-Switch connections show a 1%-3% increase in execution time. For INT8, execution time increases by 66% and 76% for double Buffer and single Buffer respectively. Near and random Via-Switch connections show an 16%-80% increase in execution time. The use of maximum possible AIEs does not enable Via-Switch (Far) connections, since most of the AIEs in the array are occupied. Thus, our experiments conclude that cascade connection demonstrates the lowest latency for both FP32 and INT8 scenarios. The rest of the experiments in this paper use Cascade connections.

Summary on choosing AIE to AIE communication interface: For low-latency communications, users can place kernels in neighboring AIEs and utilize either cascade connections

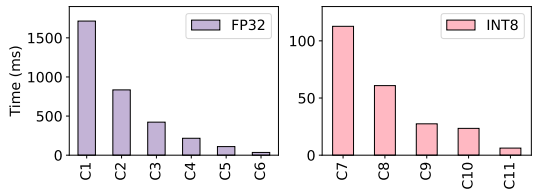

Fig. 9: Strong scaling performance analysis. Workload size used is 4096x4096x4096 for each configuration. Results obtained using hardware runs.

for streaming needs or buffer connections for non-streaming needs. Communications that can tolerate higher latencies can relax placement constraints and use switches to communicate with far-off AIEs using streams. (Figure 8)

For the next set of experiments, we define a set of hardware configurations with implementations based on CHARM. Each configuration has a unique number of AIEs, which determines the number of PLIOs used in the design and the resulting native size. Each configuration uses 32x32x32 and 64x64x64 kernel sizes for FP32 and INT8, respectively, due to its high kernel efficiency, scalability and good overlap between compute and communication as explained in Section V-C. All AIE to AIE communication is carried out using the cascade interface due to its lowest communication latency, as shown in Section V-D. All kernels use intrinsics because of performance benefits as observed in Section V-B. We have six configurations for FP32 and five configurations for INT8, as shown in Table II. All configurations use 4r2w DDR port setup to support 34 GB/s BW.

TABLE II: Hardware configurations involving multiple AIEs

Configuration	Precision	# AIEs	Native Size	# PLIOs
C1	FP32	16	32x128x128	7
C2	FP32	32	64x128x128	10
C3	FP32	64	128x128x128	20
C4	FP32	128	128x256x128	36
C5	FP32	256	256x128x256	64
C6	FP32	384	384x128x256	96
C7	INT8	16	128x256x128	14
C8	INT8	32	128x256x256	20
C9	INT8	64	256x256x256	40
C10	INT8	128	256x512x256	72
C11	INT8	256	256x512x512	112

E. AIE strong scaling analysis

Figure 9 shows the performance scaling for a workload size of 4096x4096x4096 for various configurations of the AIEs. The execution time shown is measured on hardware. The graph depicts a strong scaling scenario in which the workload remains the same but the number of AIEs increases from left to right. The execution latency decreases exponentially from left to right. The same effect is seen with an INT8 workload as well.

F. AIE weak scaling analysis

Figure 10 shows execution time as the number of AIEs increases. The workload here scales with the number of AIEs.

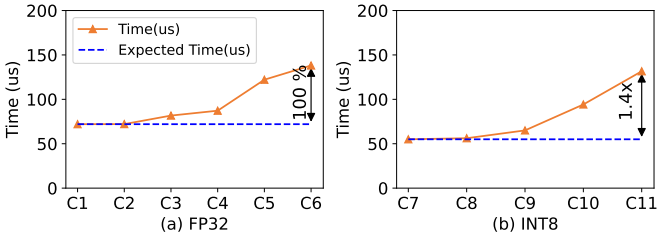

Fig. 10: Weak scaling performance analysis. Workload size is the same as native size for each configuration. Results obtained using hardware runs.

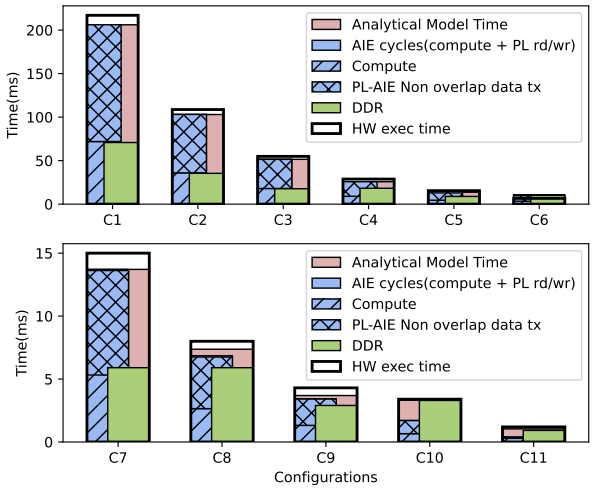

Fig. 11: Analysis of execution times for different hardware setups. A breakdown chart can reveal the specific performance limitations for each configuration. Workload size = 2048x2048x2048. (Top = FP32, Bottom = INT8). Results obtained using analytical model and hardware execution.

The execution time increases for configurations towards the right for both FP32 and INT8. The maximum difference for FP32 and int8 is 100 % and 1.4x, respectively. The time consumed by reading and writing back results increases with workload size. This observed rise in overall time can be attributed to the increased memory transactions while the computation time remains constant.

G. Multi AIE execution breakdown analysis

We extensively analyze the communication and computing aspects of the Versal architecture for various workload sizes mapped to multiple AIEs. Figure 11 shows the execution time and breakdown for a workload size of 2048x2048x2048 in various configurations for both FP32 and INT8 precision. As the number of AIEs increases, a strong scaling behavior is observed, similar to Figure 9. The observed HW execution time is close to the time reported by our analytical model for different configurations. The breakdown consists of the time consumed by DRAM communication (green), the PL and AIE communication, and the computation on the AIE. The AIE cycles (blue) consist of AIE compute cycles and exposed communication cycles between PL and AIE. Even though the AIE compute and data transfer between PL and AIE is overlapped in the kernel implementation, there is some

exposed non-overlapping time spent in data transfer from PL to AIE. This overhead is repeated once for each DRAM tile transfer from PL to AIE. This makes it directly proportional to the number of tiles in DRAM. As the configuration changes from left to right, the DRAM tiles are reduced, indirectly reducing the non-overlapping overhead. As we move to the right from configuration C4, the computing power of the configurations increases significantly. As a result, DRAM to PL transfer (green) dominates the overall time, making the workload memory bound.

The DRAM to PL transfer time is also significantly impacted by double buffering. Double buffering increases the BRAM requirement by 2×, while single buffering stores a larger input/output matrix within the same BRAM space. Thus, using single buffering can indirectly reduce the tiling overhead on DRAM while serializing the DRAM to PL with PL to AIE communication. Since all of our designs use double buffering, we briefly studied the effect of using single buffering instead of double buffering. We evaluate both FP32 and INT8 precisions using multiple AIE configurations. For FP32, single buffering increases the total execution time for all configurations. For example, in configuration C6, the time increases from 9.95 ms (double buffering) to 14.72 ms (single buffering). Here, the AIE compute time is equal to the DRAM to PL transfer time, so single buffering serializes the DRAM to PL transfers, adding more latency. However, for INT8, we observe that some configurations did indeed do better with single buffering. For example, the time reduces from 0.92 ms (double buffering) to 0.77 ms (single buffering) for C11.

Summary on using multiple AIEs: Using the maximum number of AI engines for a problem may not always lead to better performance. The off-chip memory bandwidth (DRAM BW) and the on-chip communication bandwidth (PLIO BW) can limit performance as shown in Figure 11. Different AIEs can run different kernels in parallel, unlike SMs in a GPU. Thus, we suggest utilizing these AIEs for operations that do not require external data (from PL or DRAM). Operations such as activation functions (ReLU), softmax, and elementwise addition can be performed on the output of AIEs running GEMM operations by implementing kernels in unused AIEs, instead of implementing them in the PL. This approach avoids unnecessary data movement between AIE and PL or DRAM, improving overall performance. The decision between using single buffering versus double buffering should be made on the basis of the AIE compute time and the DRAM to PL transfer time. Single buffering is advisable exclusively when the DRAM to PL time considerably exceeds the AIE compute time.

H. Effect of PLIO on performance

PLIO facilitates data transfer between PL and AIEs. Section IV-A delves into communication methods, such as packet switching and circuit switching. Packet switching facilitates communication with multiple AIEs, but it does so in a sequential manner, whereas circuit switching is only applicable for data broadcasting. In this experiment, we analyze the

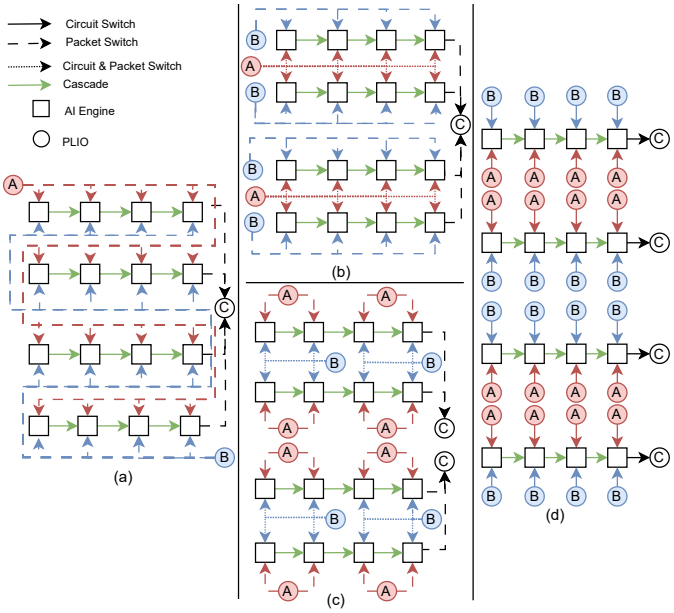

Fig. 12: Four of the 12 schemes used for understanding the effect of PLIOs on performance. The design employs 16 AIEs across all schemes. Scheme (a) exhibits the lowest PLIO utilization, resulting in poor performance. Schemes (b) and (c) achieve a balance between PLIO usage and performance, while scheme (d) demonstrates maximum PLIO utilization.

impact of PLIOs on GEMM performance. Every single PLIO port is configured with a bus width of 128-bit. We fix the number of AIEs to 16 and vary PLIO usage from 3 to 36 for FP32 and 3 to 34 for INT8. Figure 13 illustrates the impact of varying PLIOs for configurations C1 (FP32) and C7 (INT8). For each of the twelve PLIO count values (shown on the x-axis), different connectivity schemes such as packet switching, circuit switching, or a mix are used. In Figure 13 (left), the scheme with 3 PLIOs uses only packet switching, and the scheme with 36 PLIOs uses only circuit switching. The GEMM latency reduces by 4.6x for circuit switching at the cost of 12x increase in PLIOs. It also shows the effect of PLIOs on the scalability of the design when extended to the full AIE array. The 36 PLIO scheme for a 16 AIE design can only be replicated 7 times before exhausting the available PLIOs (7*36 = 252). This results in a overall AIE array utilization of 28% (7*16 = 112 AIEs) leaving 288 AIEs unused. In contrast, a 7 PLIO scheme for the same 16 AIE design can be replicated 25 times using 175 PLIOs using all 400 AIEs (100% utilization). Similar conclusions can be derived for the INT8 designs (Figure 13 (right)). Figure 12 shows four schemes out of these twelve schemes.

Figure 12(a) contains **three PLIOs**: two for input matrix A and B and one for output matrix C. Here, packet switching is the only communication method that can meet the GEMM application's requirements. The 16th AIE has to wait 16 time steps, resulting in the longest execution time for this scheme.

Figure 12(b) contains **seven PLIOs**: two for input matrix A, four for input matrix B, and one for output matrix C. Input matrix A uses a combination of circuit and packet switching

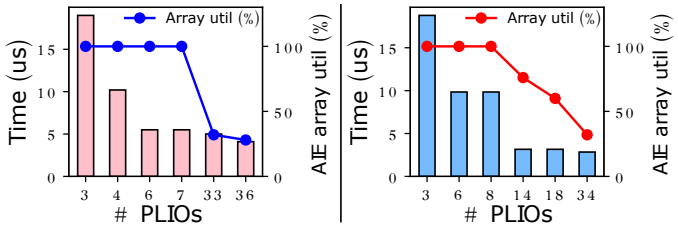

Fig. 13: GEMM performance sensitivity to number of PLIOs and the achievable AIE array utilization. (Left = FP32, Right = INT8). Left y-axis represents execution time for 16 AIEs running a native workload size of 32x128x128 and 128x256x128 for FP32 and INT8 respectively. Right y-axis shows the maximum possible utilization of whole AIE array for each scheme. For FP32 (Left) and INT8 (Right), 7 PLIOs and 14 PLIOs provide the optimal balance between resource efficiency and performance respectively. Results obtained using `aiesimulator`.

because the rows of matrix A can be reused for the columns in matrix B, enabling broadcasting or circuit switching, whereas the data across the reduction axis is packet switched. Input matrix B uses four PLIOs representing different columns of B, and the reduction axis is packet switched across the AIEs. This approach overlaps compute and data transfer across AIEs for all discussed FP32 configurations.

Figure 12(c) contains **14 PLIOs**: eight for input matrix A, four for input matrix B, and two for output matrix C. Input matrix A has eight PLIOs that each packet switch between two AIEs. Input matrix B uses four PLIOs that combine packet switching and circuit switching. Compared to (b), this scheme takes advantage of reuse in matrix B instead of matrix A. This scheme applies to all INT8 experiments in the previous section. As INT8 is 16 times faster than FP32, more PLIOs are needed to balance compute and data transfer effectively.

Figure 12(d) contains **36 PLIOs**: 16 for the input matrix A, 16 for the input matrix B and 4 for the output matrix C. Each AIE has its own PLIO. High PLIO utilization allows for higher parallel data transfer, showing the best performance.

Thus, our experiment shows that as we move from three PLIOs to 36 PLIOs, performance improves by **4.63x** for **FP32** and **6.60x** for **INT8**.

Summary on using PLIO: The amount of PLIO used can dictate the performance of AIE kernels. Adding more PLIOs yields diminishing returns (Figure 13). More AIE usage will typically mean more PLIO usage. Therefore, high PLIO usage per AIE can lead to unutilized AIEs.

TABLE III: Selected GEMM workloads from popular DNNs

Workloads	M	K	N	ID
BERT	3072	4096	1024	B1
ViT	3072	1024	4096	V1
Llama2-13B	13824	5120	4096	L1
Llama2-34B	6656	20480	4096	L2
	8192	128	3584	L3
Llama2-70B	4000	256	8192	L4

I. Analysis of real-world workloads

We analyze popular workload shapes and sizes from state-of-the-art DNNs such as Llama and BERT. Table III lists

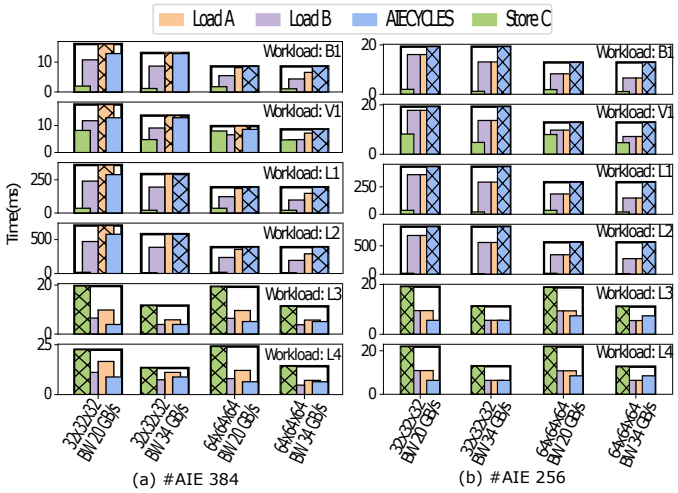

Fig. 14: Effect of changing AIE kernel size, DRAM bandwidth, and number of AIEs on various real-world workloads. Results obtained using analytical model.

some shapes and sizes of GEMM from these networks. We can observe that these workloads are not square, but they are tall, fat, or skinny. We analyze the workloads in Table III with an analytical model to obtain execution breakdown and latency. To obtain the best performance, we utilize the insights from the performance analysis in previous sections and use the most efficient kernel size for FP32 (32x32x32), with the most DRAM bandwidth (34 GB/s using 4r2w), and the C6 configuration (384 AIEs and 96 PLIOs). To demonstrate this provides the best performance, we vary the number of design ports from 2r1w to 4r2w (which varies the achieved DRAM bandwidth from 20 GB/s to 34 GB/s), vary the kernel size from 32x32x32 to 64x64x64, and vary the number of AIEs from 384 to 256 (hence, PLIOs from 96 to 64). Figure 14 shows the results. The hatched bars indicate the bottleneck.

In workloads B1, V1, L1, and L2, the initial constraint is caused by A matrix load that points to a DRAM bottleneck. By enhancing the DRAM bandwidth to 34 GB/s, the limitation shifts from load A access to PL-AIE execution cycles. Similarly, workloads L3 and L4 are constrained by the output C matrix store operation. This is attributed to the big M and N dimension and small K dimension. This is caused by the limited DRAM bandwidth. Higher bandwidth reduces execution time but not the primary bottleneck.

J. Roofline model

Figure 15 illustrates the roofline plot for AMD Versal VCK5000 for INT8 precision. We draw multiple max compute throughput lines - one for each configuration from Table II because each configuration has different number of AIEs resulting in different peak throughput. PL compute (from DSPs and CLBs) is not considered for this plot. We show two distinct bandwidth (BW) limits: one corresponding to DRAM BW and the other corresponding to the PLIO BW.

The workloads shown in Table III are plotted on the roofline. Looking at the red dots, we observe that the workloads from BERT, ViT and Llama2 (13B,34B) are compute bound as they

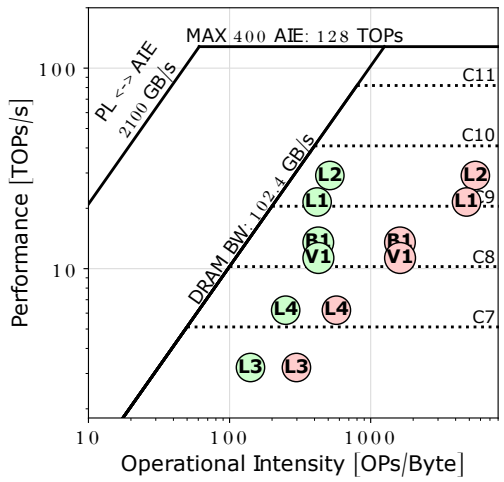

Fig. 15: Roofline plot for workloads in Table III. All executions use 4r2w DDR port setup for each workload. Results obtained using hardware runs.

appear to the right of the ridge point, whereas the workloads from Llama2-70B (L3, L4) are DRAM BW bound.

Although the VCK5000 features an aggregate internal PL memory of 24MB, including both BRAM and URAM, the effective on-chip storage capacity is lower. To keep AIEs fed and busy, the available BRAM ports need to be maximized to achieve high BRAM bandwidth, resulting in data being distributed across numerous BRAMs, each of which may be underutilized. Furthermore, implementing double buffering in the PL to overlap AIE computation with PL-to-AIE data transfer increases the BRAM requirement. Consequently, storing the entirety of a 24MB input on-chip is impractical, necessitating DRAM data tiling. Tiling results in increased memory transfers, which reduces the workload's operational intensity, pushing the workload towards the left in the roofline plot. The green circles show the same workloads when tiling overhead is included, making all of them DRAM BW bound. Hence, while the theoretical maximum compute throughput ceiling is 128 TOPS, this performance is unattainable for these workloads due to the limited DRAM BW.

Another key observation from the roofline plot is that the higher PLIO BW cannot be fully utilized due to the limited DRAM BW. However, leveraging the PL memory as local storage can help mitigate some of the limitations imposed by the restricted DRAM BW. To take full advantage of the internal PLIO BW, the entire application must fit within the PL memory, a scenario only achievable for smaller workloads.

K. Discussion: Adaptability to 2^{nd} generation Versal devices

AMD Versal AIE-ML is the second generation of AI Engines optimized for machine learning workloads. Our qualitative analysis maintains its validity and relevance for the AIE-ML architecture. However, the quantitative results are expected to change due to the enhanced features of AIE-ML, including increased AIE compute throughput, larger internal memory, and improved AIE-AIE bandwidth. Similarly, the analysis methodology developed in this work can be readily applied by other researchers to conduct similar analyses on AIE-ML. Our analytical model can also be easily adjusted to be used for AIE-ML.

VI. Conclusion

This paper presents a thorough examination of AMD Versal's performance for GEMM workloads. We pose a set of research questions related to the performance of GEMM workloads and set up experiments to answer those questions. We use SOTA implementations and their variations to perform this analysis. We offer a comprehensive set of insights derived from our analysis to assist developers in crafting efficient designs.

Our analytical model can be used by researchers and developers to quickly estimate performance. This model, along with our code and experimental setup, is available for use at: https://github.com/kmhatre14/GEMM_Performance_Analysis_on_AMD_Versal_VCK5000.

References

[1] "NVIDIA TENSOR CORES," 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/tensor-cores/

[2] "AI Engine API User Guide." 2022.

[3] AMD/Xilinx, "Versal ACAP AI Engine Architecture Manual (AM009)." 2021.

[4] ——, "AI Engine Intrinsic User Guide." 2022.

[5] ——, "AI Engine Kernel and Graph Programming Guide (UG1079)." 2022.

[6] ——, "Versal ACAP AI Engine Programming Environment User Guide (UG1076)." 2022.

[7] N. Brown, "Exploring the versal ai engines for accelerating stencil-based atmospheric advection simulation," 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. 91–97. [Online]. Available: https://doi.org/10.1145/3543622.3573047

[8] 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.

[9] P. Chen, P. Manjunath, S. Wijeratne, B. Zhang, and V. Prasanna, "Exploiting On-Chip Heterogeneity of Versal Architecture for GNN Inference Acceleration," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. Gothenburg, Sweden: IEEE, Sep. 2023, pp. 219–227. [Online]. Available: https://ieeexplore.ieee.org/document/10296434/

[10] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versaltm architecture," 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. 84–93. [Online]. Available: https://doi.org/10.1145/3289602.3293906

[11] N. P. e. a. Jouppi, "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

[12] N. Perryman, C. Wilson, and A. George, "Evaluation of xilinx versal architecture for next-gen edge computing in space," in *2023 IEEE Aerospace Conference*, 2023, pp. 1–11.

[13] G. Singh, A. Khodamoradi, K. Denolf, J. Lo, J. Gómez-Luna, J. Melber, A. Bisca, H. Corporaal, and O. Mutlu, "SPARTA: Spatial Acceleration for Efficient and Scalable Horizontal Diffusion Weather Stencil Computation," in *ICS*, 2023.

[14] E. Taka, A. Arora, K.-C. Wu, and D. Marculescu, "MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine," Nov. 2023, arXiv:2311.04980 [cs]. [Online]. Available: http://arxiv.org/abs/2311.04980

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[16] M. Wierse, "Evaluation of xilinx versal device," Bachelor Thesis, ETH Zurich, Zurich, 2023-02.

[17] 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," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. San Francisco, CA, USA: IEEE, Oct. 2023, pp. 1–9. [Online]. Available: https://ieeexplore.ieee.org/document/10323754/

[18] 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)*, Aug. 2022, pp. 200–208, iSSN: 1946-1488. [Online]. Available: https://ieeexplore.ieee.org/document/10035160

[19] 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: C omposing H eterogeneous A ccele R ators for M atrix Multiply on Versal ACAP Architecture," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey CA USA: ACM, Feb. 2023, pp. 153–164. [Online]. Available: https://dl.acm.org/doi/10.1145/3543622.3573210

[20] J. Zhuang, J. Lau, H. Ye, Z. Yang, S. Ji, J. Lo, K. Denolf, S. Neuendorffer, A. Jones, J. Hu, Y. Shi, D. Chen, J. Cong, and P. Zhou, "Charm 2.0: Composing heterogeneous accelerators for deep learning on versal acap architecture," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, no. 3, Sep. 2024. [Online]. Available: https://doi.org/10.1145/3686163

[21] J. Zhuang, Z. Yang, S. Ji, H. Huang, A. K. Jones, J. Hu, Y. Shi, and P. Zhou, "SSR: Spatial Sequential Hybrid Architecture for Latency Throughput Tradeoff in Transformer Acceleration," Feb. 2024, arXiv:2401.10417 [cs]. [Online]. Available: http://arxiv.org/abs/2401.10417

[22] J. Zhuang, Z. Yang, and P. Zhou, "AutoMM: Energy-Efficient Multi-Data-Type Matrix Multiply Design on Heterogeneous Programmable System-on-Chip," May 2023, arXiv:2305.18698 [cs]. [Online]. Available: http://arxiv.org/abs/2305.18698