

WideSA: A High Array Utilization Mapping Scheme for Uniform Recurrences on ACAP

Tuo Dai, Bizhao Shi, Guojie Luo

{daitoto, bshi, gluo}@pku.edu.cn

School of Computer Science, Peking University

Center for Energy-efficient Computing and Applications, Peking University

Abstract—The Versal Adaptive Compute Acceleration Platform (ACAP) is a new architecture that combines AI Engines (AIEs) with reconfigurable fabric. This architecture offers significant acceleration potential for uniform recurrences in various domains, such as deep learning, high-performance computation, and signal processing. However, efficiently mapping these computations onto the Versal ACAP architecture while achieving high utilization of AIEs poses a challenge.

To address this issue, we propose a mapping scheme called WideSA, which aims to accelerate uniform recurrences on the Versal ACAP architecture by leveraging the features of both the hardware and the computations. Considering the array architecture of AIEs, our approach utilizes space-time transformations based on the polyhedral model to generate legally optimized systolic array mappings. Concurrently, we have developed a routing-aware PLIO assignment algorithm tailored for communication on the AIE array, and the algorithm aims at successful compilation while maximizing array utilization. Furthermore, we introduce an automatic mapping framework. This framework is designed to generate the corresponding executable code for uniform recurrences, which encompasses the AIE kernel program, programmable logic bitstreams, and the host program. The experimental results validate the effectiveness of our mapping scheme. Specifically, when applying our scheme to matrix multiplication computations on the VCK5000 board, we achieve a throughput of 4.15TOPS on float data type, which is $1.11\times$ higher compared to the state-of-the-art accelerator on the Versal ACAP architecture.

Index Terms—Mapping, Re-configurable Array Architecture, Versal ACAP

I. INTRODUCTION

Modern heterogeneous FPGA architectures, like AMD/Xilinx Versal Adaptive Compute Acceleration Platform (ACAP) [1], combine AI Engines (AIEs) with programmable logic (PL) to boost applications in the AI and intelligent signal processing domains. In these domains, uniform recurrences [2], which comprise nested loops with uniform dependencies, are prevalent types of computations. Regrettably, there is currently a lack of established development methodologies for efficiently mapping large-scale uniform recurrences onto the Versal ACAP architecture with high utilization of AI Engines.

The ACAP architecture comprise an array of several hundred AIE cores, such as 8×50 in the VC1902 architecture [3], interconnected through a mesh network-on-chip (NoC). Each AIE core consists of vector processing and load/store units, functioning as a very-long-instruction-word (VLIW) [4] processor to deliver high-performance vectorized computations.

To facilitate communication among the AIE cores, the NoC is utilized for inter-core communication, enabling efficient data transfers between cores. Moreover, neighboring cores utilize shared buffers, providing higher bandwidth for data exchange. When it comes to data transfer to and from the AIEs, there are hundreds of I/O ports available, supporting terabytes of bandwidth.

As ACAP demonstrates a remarkable capacity for intense computation, developing acceleration designs on the architecture has become an urgent trend in recent times. However, current efforts have not succeeded in achieving high utilization of the AIE array. For example, Vitis-AI [5] introduces the DPU [6] for the VC1902 architecture, but only accomplishes a 64% AIE utilization. There are several ongoing challenges associated with developing designs with high array utilization on the Versal ACAP architecture:

- **Increased programming complexity:** Higher AIE utilization results in more cores that need to be programmed with certain intrinsics. In some situations, different cores execute different programs, necessitating significant human effort to develop such accelerators.
- **Increased placement and routing difficulty:** Mapping computations onto the Versal ACAP architecture with high utilization of AIEs often necessitates careful placement and routing of AIEs and data communications. From the perspective of AIE compilation, attaining high AIE utilization typically results in difficulties in placing cores and buffers, as well as routing streaming communications on the NoC. For example, CHARM [7] struggles to compile large designs on Vitis 2022.1.
- **Extended compilation time:** The default compilation tools provided by AMD/Xilinx Vitis employ ILP algorithms to find placement and routing solutions. Consequently, a larger number of cores results in a longer time to find a legal solution.

To address these challenges, we propose WideSA, a high array utilization mapping scheme for uniform recurrences on the Versal ACAP architecture. By leveraging the AIE array architecture, we apply space-time transformation and loop nest transformation using the polyhedral model, generating systolic-like mappings on the AIE array. On one hand, systolic designs assign similar workloads to different cores, enabling us to reuse a single core program and thereby reduce human

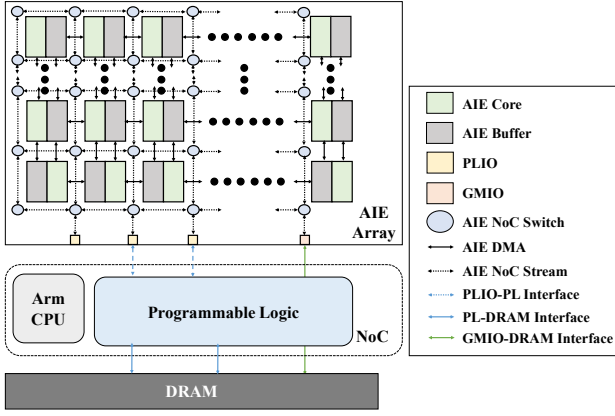


Fig. 1: Versal ACAP Architecture

effort. On the other hand, systolic designs regularize both the placement and communication of cores, simplifying the placement and routing process. Additionally, we designed a routing-aware PLIO assignment algorithm to improve the success rate of compilation. We also developed an automatic framework to generate the corresponding code for heterogeneous backends, including AIEs, PL, and host. In the evaluation section, we demonstrate the effectiveness of WideSA by successfully implementing executable acceleration systems for various uniform recurrences, accommodating different data types. Our approach achieves high throughput with high utilization of AIEs.

We summarize our contributions as follows:

- We propose a mapping scheme, based on the polyhedral model, for uniform recurrences that generates a systolic design on ACAP with high AIE utilization.
- We design a routing-aware PLIO assignment algorithm that takes into account the characteristics of systolic mappings and the AIE architecture, thereby facilitating an efficient compilation process.
- We develop an automatic framework that generates corresponding code for heterogeneous backends based on the mapping results.
- We achieve high throughput across different computations and data types, outperforming state-of-the-art methods.

II. BACKGROUND

A. Versal ACAP Architecture and Workflow

1) *Hardware Features:* AMD/Xilinx has developed the Versal ACAP architecture to cater to the increasing demands of next-generation wireless processing and machine learning applications. Figure 1 illustrates the detailed architecture of VCK5000, an evaluation kit for Versal ACAP, comprising the CPU, PL, and AIE components. The AIE array on VCK5000 consists of 8×50 AIE cores, with each core capable of generating 128 MACs of int8 data type every cycle at a frequency of 1 GHz or higher. Moreover, the AIE cores operate in single-instruction-multiple-data (SIMD) mode using

TABLE I: Different Data Communication Bandwidth on the Versal ACAP Architecture

Methods	Frequency	Bitwidth	Channels	Total
AIE DMA	1.25 GHz	256 bits	400	15.6 TB/s
AIE NoC Stream	1.25 GHz	32 bits	400	1.95 TB/s
PLIO-PL	1.25 GHz	128 bits	78	1.52 TB/s
GMIO-DRAM	1.25 GHz	64 bits	16	0.125 TB/s
PL-DRAM	0.50 GHz	-	4	0.100 TB/s

a VLIW pattern, enabling acceleration of a large number of vectorized computations.

In Figure 1, we identify five data transfer methods, including those within the AIE array and among the AIE, PL, and DRAM components. These methods are referred to as **AIE DMA**, **AIE NoC stream**, **PLIO-PL**, **PL-DRAM**, and **GMIO-DRAM** interfaces. We profile these data transfer methods on VCK5000 and present the results in Table I. Within the AIE array, each AIE core has direct memory access (DMA) ports connected to four neighboring local buffers with a width of 256 bits. Using the **AIE DMA** method, a total data transfer rate of up to 15.6 TB/s can be achieved. Furthermore, each AIE core is linked to the NoC through a stream interface with a width of 32 bits. The data transfer bandwidth through the **AIE NoC stream** method reaches a maximum of 2 TB/s, which is lower compared to the DMA method. The PLIO ports, responsible for data communication between the PL and AIE array, can provide a maximum bandwidth of 1.52 TB/s. Based on the profiling results, utilizing the **AIE DMA** method for data transfer proves beneficial in overcoming communication bottlenecks, aligning with the dataflow in systolic array designs. In terms of data communication with DRAM, the bandwidth is approximately 0.1 TB/s, significantly lower than the on-chip data transfer methods. This observation inspires us to exploit data locality within computations to enhance overall performance.

2) *Software Programming Model:* AMD/Xilinx offers a development tool for AIEs and Versal ACAP integrated into Vitis. The programming model [8] designed for AIEs consists of two levels: a graph program across the AIE array with each node representing an AIE kernel program. The graph program represents the dataflow information among AIE kernels and between the AIE and I/O ports. The compiler in Vitis transforms the dataflow graph into a subnetwork of physical AIE cores, determines the placement of buffers, and configures NoC stream routing. Since placement and routing are NP-hard problems, the compiler employs ILP solvers to process these two phases. However, as the design scale increases and AIE utilization becomes high, finding a legal solution efficiently becomes challenging for the solvers [9]. To address this, incorporating constraints for placement and routing helps alleviate the congestions and accelerates the solvers in finding solutions. The systolic design scheme provides a regular pattern for placement and routing, which is suitable for constructing these constraints.

B. Uniform Recurrences and Systolic Array Mapping

Uniform recurrences refer to computations that consist of nested loops, where all dependencies are uniform. These types of computations are commonly found in AI and signal processing applications, such as matrix multiplication, 2D convolution, FIR filtering, and so on. Several prior works [10]–[12] have focused on generating systolic array designs for uniform recurrences on FPGAs, employing the polyhedral model for loop transformations to explore successful mappings. The polyhedral model [13], [14] serves as a compilation framework for loop transformation, encompassing space-time transformation, latency hiding, SIMD vectorization, fusion, and more. A legal combination of these transformations represents a schedule within the polyhedral model, and the goal of systolic design mapping is to find the optimal schedule.

An AIE kernel handles more computations compared to a PE in typical systolic arrays. Additionally, specific hardware features of the AIE array differ from those of common systolic arrays. As a result, the mapping problem on the Versal ACAP architecture is not a straightforward systolic array mapping. Consequently, it is necessary to model corresponding transformations and constraints within the polyhedral model, an area that has not yet been extensively researched.

III. SYSTOLIC MAPPING SCHEME ON ACAP

A. Kernel Scope Demarcation

According to the programming model of AIEs, it is necessary to demarcate the scope of codes mapped to execute on a single AIE core and the outer loop nests to be mapped to the AIE array. This demarcation allows us to decompose the mapping problem into graph-level mapping and kernel-level mapping, which are independent of each other after selecting tiling factors.

Polygonal tiling [15], [16], an effective solution for workload partitioning in uniform polyhedral domains, plays a crucial role in determining the innermost and outer loop nests for tiling. We illustrate the tiling process using the MM example with (N_0, M_0, K_0) as the tiling factors, involving loop re-indexing, tiling, and rewriting, as depicted in Figure 2. Building on prior works, we consider the specific features of the AIE array when performing the demarcation.

B. Systolic Mapping Generation

To generate systolic array designs on the AIE array following kernel scope demarcation, we utilize the polyhedral

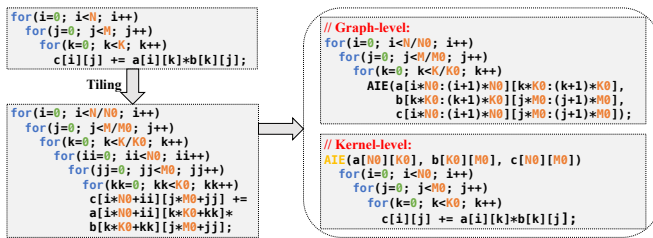


Fig. 2: Kernel Scope Demarcation

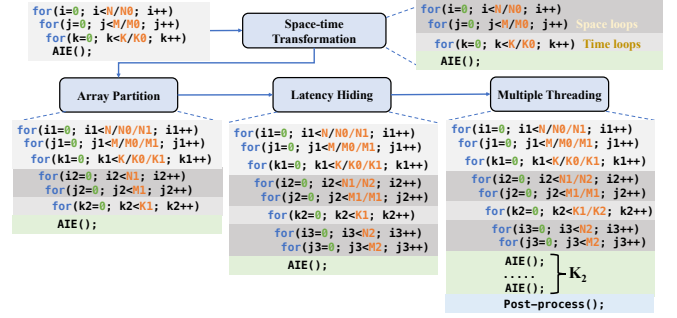


Fig. 3: Polyhedral Model-Based Systolic Mapping

model, drawing inspiration from AutoSA [11], to facilitate loop transformations. To be specific, we employ four types of transformation techniques, as depicted in Figure 3.

1) *Space-time Transformation*: The first step involves performing space-time transformation to map the graph-level loop nests to a systolic array design. We identify loops in the outermost loop band with dependence distances no greater than one and consider them as candidate space loops. Subsequently, we enumerate all possible combinations of space loops from the candidate pool. The selected space loops are then permuted in the outermost position, while the loops below them are designated as time loops. Due to the constraints imposed by the hardware shape of the AIE array, the mapper generates only 1D and 2D systolic arrays. This step results in the generation of multiple systolic arrays, each with a unique schedule. As shown in Figure 3, we choose loops i and j as the space loops (on dark gray background) and loop k as the time loop (on light gray background) in the MM example.

2) *Array Partition*: To accommodate the limited number of AIEs in the horizontal and vertical directions of the AIE array, array partitioning becomes necessary when mapping a large array. In order to achieve this, we apply tiling to the outermost permutable loop that contains the space loops. In Figure 3, we illustrate an example where we tile the outermost loop band in the MM example using the tiling factors (N_1, M_1, K_1) . The point loops originating from the original loops are retained as the space loops. This results in a 2D systolic array with dimensions of $N_1 \times M_1$ (on dark gray background).

3) *Latency Hiding*: Latency hiding plays a crucial role in mitigating the pipeline stalls caused by loop-carried dependencies in computational statements. In the case of the MM example, the accumulate operations in the statement introduce loop-carried dependence within the loop, resulting in long latency in the systolic chain. To address this issue, we identify parallel loops in the polyhedral model schedules, applies tiling to these loops, and permutes the point loops to the innermost position. As an illustration, loops i and j are identified as parallel loops in the MM example. We extract them using the tiling factors (N_2, M_2) and permute the point loops to the innermost position. Since there are no loop-carried dependencies on the innermost loop, the latency of design reduce as the chain length shortened.

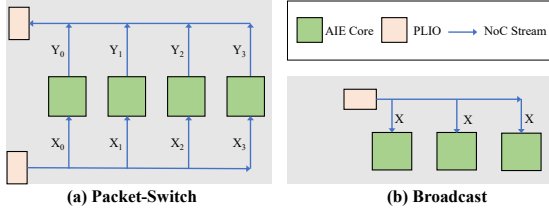


Fig. 4: Communication Methods for PLIO Ports Utilization Reduction

4) *Multiple Threading*: As AIE cores execute concurrently, the AIE array inherently supports multiple threading. Leveraging this characteristic, utilizing multiple AIEs to execute the same instructions but different indexing can significantly enhance overall performance. We identify parallelizable loops in the time loops that do not have data dependence. In the MM example, the loop k is identified as a parallelizable loop. We can apply tiling to this loop using the factors K_2 . The point loop is permuted to the innermost position and completely unrolled to generate multiple threads of AIEs.

C. Placement and Routing Constraints Construction

The systolic design generated in the previous section represents an abstract mapping scheme. Consequently, it is essential to use the space loops as input and generate an actual mapped graph for AIE array that considers placement and routing constraints. The mapped graph consists of nodes, representing AIE cores and input/output ports, and edges, which connect the ports of the nodes. The placement and routing constraints involve assigning coordinates to the AIE cores, buffers, and input/output ports, as well as determining the routing paths for the edges. In the subsequent subsections, we introduce the graph builder and routing-aware PLIO assignment, which are responsible for constructing the mapped graph and generating the associated placement and routing constraints, respectively.

1) *Graph Builder*: To construct the mapped graph, we iterate through all coordinates in the space loops and create a node for each pair of coordinates in the 2D systolic array, representing an AIE core. Next, we identify the data communications between AIE cores based on the dependencies within the space loops. Following the definitions in AutoSA [11], there are three types of data dependences:

- Read dependence: Transfer the read-only data.
- Flow dependence: Transfer the intermediate data.
- Output dependence: Transfer the output-only data.

Based on these data dependences and the space loops, we define the I/O ports and edge directions. Since AIEs do not support intermediate results between different iterations, we treat flow dependences as input dependencies when constructing I/O ports. The polyhedral model for the array access to matrix A in the MM recurrences is $\{i, j, k\} \rightarrow \{i, j+1, k\}$, and when loops j, k are the space loops, the direction is $(1, 0)$. We connect the input ports from the corresponding nodes with a constant and non-zero distant direction.

As for the output ports, the boundary input ports, and the zero distant direction ports, we create PLIO ports as the other end of the connection edge. To adhere to the limitation on the number of PLIO ports, we utilize packet-switch communications and broadcast communications to reduce the number of used ports, as depicted in Figure 4.

2) *Routing-Aware PLIO Assignment*: Once we have the mapped graph, we search for available cores on the AIE array to place the AIE kernels. To facilitate efficient communication between neighboring cores, we assign the buffers of ports connecting these cores to the shared buffer of the cores, forming part of the placement constraints. These constraints enable the transformation of the kernels' placement into a regular duplicate pattern of a single kernel.

Aside from facilitating neighboring communication, it is necessary to construct paths between PLIO ports and AIE cores for data input and output. Considering the mesh structure of the NoC on the AIE array, and given that PLIOs are always located in Row 0, we can compute the routing congestion by counting the horizontal data transfer numbers. For instance, we compute the congestion for the *west* direction as follows:

$$Cong_i^{west} = \sum_{p \in \text{PLIOs}, x \in \text{AIEs}} W_i[p][x],$$

$$W_i[p][x] = \begin{cases} 1 & (p_{col} < i \text{ and } x_{col} > i \text{ and } (x, p) \in \text{Edges}) \text{ or} \\ & (p_{col} > i \text{ and } x_{col} < i \text{ and } (p, x) \in \text{Edges}) \\ 0 & \text{Otherwise} \end{cases}$$

where p_{col} and x_{col} represent the column coordinates of PLIO p and AIE x , respectively.

The computation of the congestion for the *east* direction is symmetrical.

Consequently, the routing challenges essentially transform into issues of PLIO assignment. We formulate the assignment of PLIO ports as a satisfiability problem subject to routing resource constraints. We check if there exists a set of values for PLIOs that satisfies the following constraints:

$$\forall i \in \text{Columns}, Cong_i^{west} \leq RC_{west}, \quad Cong_i^{east} \leq RC_{east}$$

where RC_{west} and RC_{east} denote the available routing resources in the AIE array.

To seek the feasible assignment of PLIO ports, we employ a heuristic greedy algorithm outlined in Algorithm 1. In this algorithm, we initialize the placement of the PLIO ports by calculating the median value of the row numbers of the connected AIE cores. If the initially computed placement coordinate is not available, we search for the nearest available coordinate instead. This heuristic greedy algorithm balances the routing congestion among the PLIO ports. By considering the connectivity with the AIE cores, it generates an optimal placement for the PLIO ports, ensuring successful routing on the NoC. The algorithm takes into account the availability of coordinates and selects the most suitable placement to minimize congestion.

By generating these constraints for the placement and routing of AIE kernels, buffers, and PLIO ports, we can

Algorithm 1 Routing-Aware PLIO Assignment Algorithm**Require:** Numbers of PLIO ports N , AIE cores X **Ensure:** Initialized PLIO assignment set P

```

1: Initialization available placement sets  $A$  as all columns
   that have PLIO ports.
2: for  $i \leftarrow 1$  to  $N$  do
3:    $S = [], num = 0$ 
4:   for  $x \in X$  do
5:     if  $(p, x) \in \text{Edges}$  then
6:        $S.append(x_{col})$ 
7:        $num++ = 1$ 
8:     end if
9:   end for
10:  Sort  $S$  to find the median:  $sort(S, S + num)$ 
11:   $P[i] = \text{find\_nearest}(A, S[num/2])$ 
12:   $remove(A, P[i])$ 
13: end for
14: return  $P$ 

```

significantly simplify the task for the AIE compiler. These constraints provide valuable information and guidelines for the compilers to optimize the placement and routing process, ultimately leading to a high utilization of the AIE array.

IV. AUTOMATIC MAPPING FRAMEWORK

To facilitate the computation of uniform recurrence, we have developed an automatic mapping framework that implements the full functional modules on the Versal ACAP architecture, as shown in Figure 5.

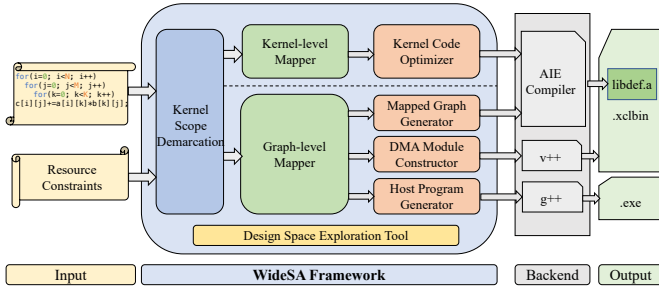


Fig. 5: Overview of WideSA Automatic Framework

Specifically, we introduce a kernel-level mapper, a DMA module constructor, and a host program generator, which work in conjunction with the kernel scope and graph mapper described in the previous section.

The kernel-level mapper and optimizer transform the C++ program into a program with AIE intrinsics, leveraging the capabilities of the AIE vector processor to exploit parallelism and optimize performance. Moreover, we design the architecture of efficient DMA modules, which serve as the buffers of AIEs on the PL, in the DMA module constructor. This architecture is tailored to the characteristics of both the hardware and the computations involved. In addition, we engineer a host program generator to generate a controller program that oversees global scheduling.

TABLE II: Evaluation Benchmarks

Benchmarks	Dimension	Problem Size	Data Types
MM	[i, j, k]	[8192, 8192, 8192]	Float
		[10240, 10240, 10240]	Int8
		[9600, 9600, 9600]	Int16
		[8192, 8192, 8192]	Int32
2D-Conv	[h, w, p, q]	[10240, 10240, 4, 4]	Float
		[10240, 10240, 8, 8]	Int8
		[10240, 10240, 4, 4]	Int16
		[10240, 10240, 4, 4]	Int32
2D-FFT	[row, col]	[8192, 8192]	Cfloat
		[8192, 8192]	Cint16
FIR Filter	[n, taps]	[1048576, 15]	Float
		[1048576, 15]	Int8
		[1048576, 15]	Int16
		[1048576, 15]	Cfloat

V. EVALUATION

A. Benchmark and Experimental Setup

In this section, we select four representative uniform recurrences with various data types as benchmarks to evaluate the performance of WideSA. The selected benchmarks include matrix multiplication (MM), 2D convolution (2D-Conv), 2D Fast Fourier Transformation (2D-FFT), and FIR filter [17]. The problem sizes and corresponding data types are provided in Table II. Here, Cfloat refers to the complex float data type and Cint16 refers to the complex 16-bit integer data type. All the experiments are conducted on VCK5000 with 250 MHz on PL and 1.25 GHz on AIE. AMD/Xilinx Vitis 2022.1 is used as the compilation backend tool.

B. Full System Performance

We conducted a comparison of the throughput between WideSA and other state-of-the-art AIE designs for the same problem size. For the MM benchmark, we successfully compiled the CHARM code [7] for the target VCK5000 with AMD/Xilinx Vitis 2022.1, incorporating placement and routing constraints, as the baseline. As for the 2D-Conv benchmark, we selected the released 8-PEs version of Vitis-AI DPU [5] which only supports Int8 data type, utilizing 256 AIEs running at 1.33 GHz and the PL at 350 MHz, as the baseline. Furthermore, we used the open-source designs from the Vitis DSP Library [18] as the baselines for the 2D-FFT and FIR filter benchmarks.

The results presented in Table III demonstrate that WideSA achieves significantly higher throughput with high utilization of AIEs. Additionally, we computed the AIE efficiency by considering the throughput and the number of used AIEs. The results indicate that WideSA maintains similar efficiency to [7] for MM, as both approaches exhibit AIE utilization over 95%. When compared to the baselines with lower AIE utilizations, WideSA trades AIE efficiency (TOPS/#AIEs) for a high overall performance (TOPS) and is bounded by memory bandwidth.

Moreover, we conducted a comparison of the performance and energy efficiency of MM using WideSA and PL-only designs on the VCK5000 target, which has 1968 DSP58 IPs at total. For the PL-only designs, we utilize AutoSA [11] as the systolic array generator. The results presented in Table IV

TABLE III: Comparison of Throughput and AIE Efficiency on Benchmarks

Method	Metric	MM				2D-Conv				2D-FFT		FIR Filter			
Data type		Float	Int8	Int16	Int32	Float	Int8	Int16	Int32	Cfloat	Cint16	Float	Int8	Int16	Cfloat
Baseline	#AIEs	384	384	384	384	-	256	-	-	10	10	10	10	10	10
	TOPS	3.73	29.78	7.82	3.72	-	31.40	-	-	0.04	0.13	0.15	2.56	0.62	0.15
	TOPS/#AIEs	0.010	0.077	0.020	0.010	-	0.123	-	-	0.004	0.013	0.015	0.256	0.062	0.015
WideSA	#AIEs	400	400	400	400	400	400	400	400	320	320	256	256	256	256
	TOPS	4.15	32.49	8.10	3.92	4.50	36.02	10.35	4.48	1.10	3.83	2.92	39.3	9.47	2.89
	TOPS/#AIEs	0.010	0.081	0.020	0.010	0.011	0.090	0.025	0.011	0.003	0.012	0.012	0.100	0.037	0.011

TABLE IV: MM Performance Comparison between PL-only and WideSA Design

Data Type	PL-only				WideSA			
	Float	Int8	Int16	Int32	Float	Int8	Int16	Int32
DSPs	1536	1528	1516	1536	152	60	67	65
#AIEs	0	0	0	0	400	400	400	400
TOPS	0.59	5.77	2.16	0.60	4.15	32.49	8.10	3.92
Power (W)	19.5	18.8	18.6	19.5	55.8	54.4	54.9	55.6
TOPS/W	0.03	0.31	0.12	0.03	0.07	0.60	0.15	0.07
Norm. TOPS/W	1.00x	1.00x	1.00x	1.00x	2.25x	1.94x	1.29x	2.25x

demonstrate that our approach achieves up to $2.25\times$ higher energy efficiency compared to the PL-only designs.

C. Scalability of WideSA on MM examples

We evaluate the scalability of WideSA while increasing AIE utilization and analyze how various factors influence performance. The results, presented in Figure 6, show a significant increase in throughput as the number of AIEs increases. In addition, the AIE efficiency results demonstrate that our approach scales effectively from small-scale to large-scale designs. However, when the number exceeds 200, the efficiency of a single AIE core decreases due to the memory-bound condition caused by the number of PLIOs and the size of the PL buffer. The increase in PLIO numbers and buffer sizes leads to increased throughput, suggesting that enhancing the bandwidth between different fabrics of ACAP can improve performance. This indicates that managing the resources and data flow between different components of the ACAP is crucial for achieving better performance.

VI. CONCLUSION

In this paper, we present a high array utilization mapping scheme for uniform recurrences on the Versal ACAP architecture. Additionally, we propose several optimizations aimed at enhancing overall performance within an automatic mapping framework. Through extensive evaluations using typical benchmarks and diverse data types, we assess the efficiency of the WideSA framework. In the future work, we aim to integrate WideSA into the MLIR-AIE workflow and develop an end-to-end compilation tool that incorporates automatic design space exploration.

ACKNOWLEDGEMENT

This work was partly supported by the National Natural Science Foundation of China (Grant No. 62090021) and the National Key R&D Program of China (Grant No. 2022YFB4500500).

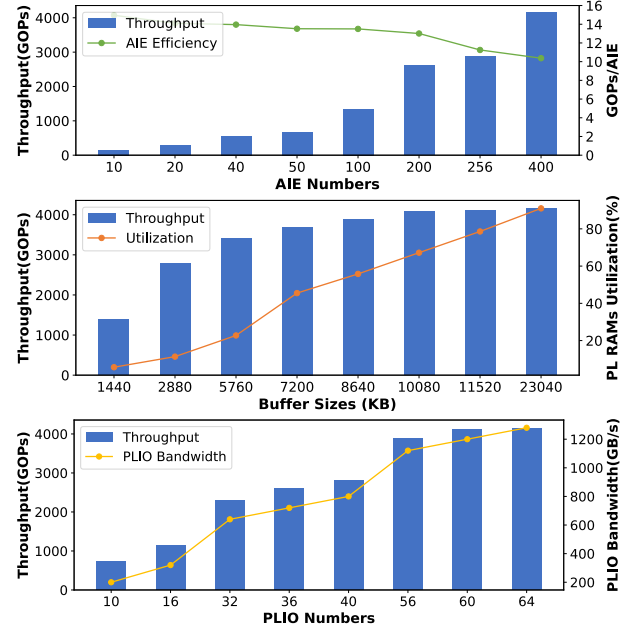


Fig. 6: Throughput Evaluation of Different AIE Numbers, PLIO Numbers, and PL Buffer Sizes

REFERENCES

- [1] AMD/Xilinx. Versal Adaptive Compute Acceleration Platform.
- [2] R. M. Karp *et al.*, "The organization of computations for uniform recurrence equations," *J. ACM*, 1967.
- [3] S. Ahmad *et al.*, "Xilinx first 7nm device: Versal AI Core (VC1902)," in *HCS*, 2019.
- [4] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *ISCA*, 1983.
- [5] Vitis AI Library User Guide. [Online]. Available: <https://xilinx.github.io/Vitis-AI/>
- [6] X. Jia *et al.*, "XVDPU: A high performance CNN accelerator on the Versal platform powered by the AI Engine," in *FPL*, 2022.
- [7] J. Zhuang *et al.*, "CHARM: Composing heterogeneous accelerators for matrix multiply on Versal ACAP architecture," in *FPGA*, 2023.
- [8] AMD/Xilinx. AI Engine Kernel and Graph Programming Guide.
- [9] W. Cook *et al.*, "An exact rational mixed-integer programming solver," in *IPCO*, 2011.
- [10] J. Cong *et al.*, "PolySA: Polyhedral-based systolic array auto-compilation," in *ICCAD*, 2018.
- [11] J. Wang *et al.*, "AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA," in *FPGA*, 2021.
- [12] Y.-H. Lai *et al.*, "SuSy: A programming model for productive construction of high-performance systolic arrays on FPGAs," in *ICCAD*, 2020.
- [13] M.-W. Benabderrahmane *et al.*, "The polyhedral model is more widely applicable than you think," in *CC*, 2010.
- [14] U. Bondhugula, "Compiling affine loop nests for distributed-memory parallel architectures," in *SC*, 2013.
- [15] R. Andonov *et al.*, "Optimal semi-oblique tiling," *IEEE TPDS*, 2003.
- [16] C. Rossetti *et al.*, "Algebraic tiling," in *IMPACT*, 2023.
- [17] K. K. Parhi, "VLSI digital signal processing systems: Design and implementation," 2007.
- [18] AMD/Xilinx. Vitis DSP Library for digital signal processing.