ARIES: An Agile MLIR-Based Compilation Flow for Reconfigurable Devices with AI Engines

Jinming Zhuang*
Brown University
Providence, USA
jinming_zhuang@brown.edu

Shaojie Xiang*
Cornell University
Ithaca, USA
sx233@cornell.edu

Hongzheng Chen
Cornell University
Ithaca, USA
hzchen@cs.cornell.edu

Niansong Zhang
Cornell University
Ithaca, USA
nz264@cornell.edu

Zhuoping Yang
Brown University
Providence, USA
zhuoping_yang@brown.edu

Tony Mao
Cornell University
Ithaca, USA
twm59@cornell.edu

Zhiru Zhang
Cornell University
Ithaca, USA
zhiruz@cornell.edu

Peipei Zhou
Brown University
Providence, USA
peipei_zhou@brown.edu

Abstract

As AI continues to grow, modern applications are becoming more data- and compute-intensive, driving the development of specialized AI chips to meet these demands. One example is AMD's AI Engine (AIE), a dedicated hardware system that includes a 2D array of high-frequency very-long instruction words (VLIW) vector processors to provide high computational throughput and reconfigurability. However, AIE's specialized architecture presents tremendous challenges in programming and compiler optimization. Existing AIE programming frameworks lack a clean abstraction to represent multi-level parallelism in AIE; programmers have to figure out the parallelism within a kernel, manually do the partition, and assign sub-tasks to different AIE cores to exploit parallelism. These significantly lower the programming productivity. Furthermore, some AIE architectures include FPGAs to provide extra flexibility, but there is no unified intermediate representation (IR) that captures these architectural differences. As a result, existing compilers can only optimize the AIE portions of the code, overlooking potential FPGA bottlenecks and leading to suboptimal performance.

To address these limitations, we introduce ARIES, an agile multi-level intermediate representation (MLIR) based compilation flow for reconfigurable devices with AIEs. ARIES introduces a novel programming model that allows users to map kernels to separate AIE cores, exploiting task- and tile-level parallelism without restructuring code. It also includes a declarative scheduling interface to explore instruction-level parallelism within each core. At the IR level, we propose a unified MLIR-based representation for AIE architectures, both with or without FPGA, facilitating holistic optimization and better portability across AIE device families. For the General Matrix Multiply (GEMM) benchmark, ARIES achieves 4.92 TFLOPS, 15.86 TOPS, and 45.94 TOPS throughput under FP32, INT16, and, INT8 data types on Versal VCK190 respectively. Compared with the state-of-the-art (SOTA) work CHARM for AIE, ARIES improves the throughput by 1.17x, 1.59x, and 1.47x correspondingly. For ResNet residual layer, ARIES achieves up to 22.58x speedup compared with optimized SOTA work Riallto on Ryzen-AI NPU. ARIES is open-sourced on GitHub: ***https://github.com/arc-research-lab/Aries***.

CCS Concepts

• **Hardware → High-level and register-transfer level synthesis**; • **Software and its engineering → Compilers**.

Keywords

Compiler, MLIR, AIE Architecture, Hardware Accelerator

ACM Reference Format:
Jinming Zhuang, Shaojie Xiang, Hongzheng Chen, Niansong Zhang, Zhuoping Yang, Tony Mao, Zhiru Zhang, and Peipei Zhou. 2025. ARIES: An Agile MLIR-Based Compilation Flow for Reconfigurable Devices with AI Engines. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '25), February 27-March 1, 2025, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3706628.3708870

1 Introduction

The rapid growth of compute- and data-intensive applications, such as artificial intelligence (AI) and wireless communications, has pushed the limits of traditional computer architectures. To address the increasing demand for computational resources, specialized chips have been developed [1, 2, 3, 4]. Among these, the AMD AI-Engine (AIE) [2] stands out as a promising solution. As shown in Figure 1, AMD AIE architecture consists of a 2D array of high-frequency VLIW processors ranging from 20 to 400 for different AIE device families. The AIE cores can communicate with each other by direct memory access (DMA) through AXI stream networks and switches. The AIE architecture adopts a multi-level scratch pad memory-based hierarchy. While L1 memory refers to the local memory within each AIE core, L2 memory represents the on-chip buffers on the PL or the memory tiles shared by all of the AIE cores. A large amount of data can be stored in the off-chip L3 memory.

AIE architecture offers a wealth of hardware resources, allowing programmers to exploit parallelism at various levels within their applications. However, efficiently mapping applications to AIE hardware remains challenging. Figure 1 demonstrates an example of mapping a Multi-Layer Perceptron (MLP) with two matrix multiplication tasks to it. To achieve higher parallelism, programmers usually exploit (1) task-level parallelism, which maps each task within the application to specific AIE core groups, and (2) tile-level

*Both authors contributed equally to this research.

FPGA '25, February 27-March 1, 2025, Monterey, CA, USA

ACM ISBN 979-8-4007-1396-5/25/02
https://doi.org/10.1145/3706628.3708870

parallelism, which partitions loop nests of a task into smaller tiles and executes these tiles in parallel across multiple AIE cores. There are also finer-grained parallelisms to be exploited inside the AIE core, including (3) loop-level parallelism which pipelines instructions inside a loop, (4) data-level parallelism which processes multiple data elements using Single Instruction Multiple Data (SIMD) vector engines, and (5) instruction-level parallelism which executes multiple instructions at a time through VLIW instruction bundling.

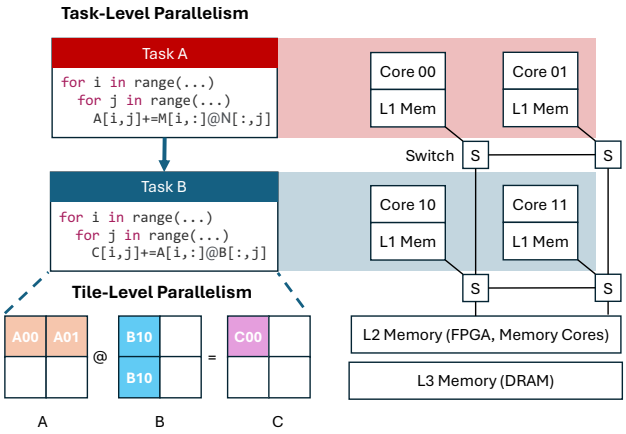


Figure 1: Mapping multi-task program to AIE cores: Memory cores are on-device memory featured in AMD AIE-ML [5].

The AIE's specialized architecture provides great opportunities for multi-level parallelism in compute-intensive tasks, but poses challenges in programming and optimization. Many recent efforts have focused on developing programming abstractions and optimization tools for AIE, aiming to enable programmers to efficiently exploit parallelism mapping. However, these proposed solutions face several limitations, hindering their widespread adoption and efficacy in fully leveraging the potential of the AIE architecture.

Challenge 1: Limited Support for Multi-Layer Applications. Recent studies on AIE architectures [6, 7, 8, 9] have focused on accelerating specific kernel, such as matrix multiplication or stencil computations. However, extending these designs to support more complex multi-layer applications often presents significant challenges even when leveraging the provided overlays. This includes tasks such as workload partitioning on heterogeneous components and inter-layer communication control, which limits the productivity of AIE architectures for multi-layer applications.

Challenge 2: Fragmented Abstraction for Multi-Level Parallelism. Some recent studies propose general programming models for AIE, such as Riallto [10] and MLIR-AIE [11]. These works mostly adopt the dataflow model as an abstraction for AIE architectures. Figure 2 shows an example in Riallto, where each kernel needs to be assigned to a specific AIE core in the 2D array (Lines 3-4). Dataflow model clearly captures task-level parallelism. However, it falls short of describing tile-level parallelism inside a task. To map a task to multiple AIE cores, programmers have to manually break down the loop tiles inside a task into separate kernels and place them to particular AIE cores; this also involves manually coordinating data transfer between loop tiles to reduce communication overheads.

To exploit intra-core parallelism, users need to insert directives into C++ code linked to each kernel (Lines 3-4) to enable loop

```
1 class MultiLayerPercepton: # Layer = 2
2     def __init__(self):
3         self.task1 = Kernel("gemm.cpp", tloc=(0,0))
4         self.task2 = Kernel("gemm.cpp", tloc=(0,1))
5     def callgraph(self, x_in, weight1, weight2):
6         out0 = self.task1(x_in, weight1)
7         return self.task2(out0, weight2)
```

Figure 2: Two-layer MLP in Riallto: The `tloc` option specifies the location of AIE core for the kernel to be executed on.

pipeline and use intrinsic C++ vector library APIs to leverage SIMD vectorization. This approach not only fragments the programming abstraction between different design levels but also complicates the exploitation of multi-level parallelism in AIE for programmers.

Challenge 3: Limited Support for Automation and Optimization. The existing compilation flows including Riallto and MLIR-AIE provide a Python-based exploration framework to utilize the feature of AIEs. However, they rely on users to provide optimized dataflow, inter-tile data movement scheme, and vectorization to achieve good performance. MLIR-AIR [12] is an automatic compilation framework for AIE architectures. However, it does not explore the customized dataflow, data types, and buffer reuse opportunities well on the FPGA side. The AMD Vitis flow uses ADF graphs to program AIE and HLS/RTL for Programmable Logic (PL). An additional configuration file is required to establish connections between PL and AIE, resulting in a fragmented programming process and difficulties in achieving holistic optimization.

Challenge 4: Portability. Various of AIE architectures exist. For example, the Ryzen-AI Neural Processing Unit (NPU) [13] is designed for consumer devices, featuring fewer AIE-ML cores along with specialized mem-tiles as `L2 memory`. In contrast, Versal ACAP [14] is geared towards high-performance computing and data centers, offering more AIE cores for greater computational power, along with an integrated FPGA for enhanced memory buffering and flexibility. Most existing works focus on either NPU or Versal, and the lack of portability between these architectures makes it labor-intensive to port designs from one platform to another.

To address these limitations, we introduce ARIES, an agile compilation flow for AIE-based reconfigurable devices. ARIES provides a novel programming model that allows programmers to define tasks at the level of tiles, with each tile handling a portion of the original problem size. Programmers have user-level control over these task tiles and can scale them out across more AIE cores without restructuring code. Additionally, ARIES offers scheduling primitives to optimize the performance of each task tile running on a single AIE core. At the IR level, ARIES introduces a unified MLIR-based representation for AIE architectures, both with and without FPGA, enabling holistic optimization of the target application. This approach enhances portability and performance across different AIE device families, including the Versal ACAP and Ryzen-AI NPU families. To summarize, our major contributions include:

- At the programming model level, ARIES introduces a novel programming abstraction that helps users exploit multi-level parallelism in AIE with more productivity; it allows programmers to define tasks at the granularity of tiles, and map task tiles across AIE cores to exploit both task- and tile-level parallelism without code restructuring. Additionally, ARIES provides a set of

declarative scheduling primitives that allows users to optimize the performance of the task tile running on a single AIE core.

- Underlying the programming interface, ARIES is the first framework providing a unified MLIR-based representation for AIE core, AIE graph, and PL by using existing AIEVec, proposed ADF, and other existing builtin dialects respectively. The unified representation enables ARIES to perform global and local optimizations. It also provides the extensibility for customized optimizations to be easily integrated to achieve near-theoretical performance. The experiments show that ARIES is capable of achieving up to 87% AIE efficiency even when scaling to hundreds of AIEs.
- ARIES demonstrates portability across different AIE architectures. With the unified IR, ARIES is able to generate low-level code for multiple AIE architectures by extending lightweight code generator backends, enabling the mapping of the same input source code to both Versal ACAP and Ryzen-NPU.

2 Background and Related Work

Accelerator Designs on AIE. Prior accelerator designs have targeted AIE architectures for specific applications. CHARM23 [6] and AutoMM [15] provide design space exploration (DSE) guided mapping solutions for matrix-multiply (MM) related applications on Versal. CHARM24 [7] further improves the performance by optimizing the intra-AIE and AIE array efficiency. MAXEVA [8] also targets dense MM acceleration on Versal. However, it focuses on the simulation of the AIE array without considering the communication optimization between PL to AIE and between off-chip memory to on-chip buffers thus lacking the real on-board performance analysis. AIM [16] makes full use of the heterogeneity of Versal ACAP to speed up the large integer multiplication-based benchmarks. SSR [17] and EQViT [18] explore the latency and throughput trade-off for transformer-based models on VCK190. HGC-N [19] design sparse and dense MM accelerators on Versal VCK5000 to accelerate the graph neural networks. SPARTA [9] leverages MLIR-AIE [11] to accelerate stencil computations on VCK190. Nevertheless, these works focus on specific applications without high extensibility and portability to other applications and platforms.

Accelerator Programming Models. Many recent works have been proposed to provide general-purpose programming models for AIE or AIE-like dataflow accelerators. Riallto [10] introduces dataflow abstraction for NPU programming. Ryzen-AI-SW [20] allows users to run AI models on NPUs by offering a pre-built AIE overlay that accelerates common operators. MLIR-AIR [12] and MLIR-AIE [11] support both ACAP and NPU, but they do not capture FPGA in ACAP architecture. Alongside AIE-specific programming tools, ML frameworks like Torch/XLA [21] support lowering Torch models to TPUs [1]. Triton [22] simplifies memory and thread management on GPUs with a tile-based abstraction and extends support to other dataflow accelerators [4]. Allo [23] introduces a programming model that allows users to apply decoupled hardware customizations [24] without changing the algorithm. These accelerator programming frameworks either fully automate compilation, hiding hardware details away, or offer user-friendly abstraction to apply hardware customization with enhanced productivity.

MLIR Compiler Infrastructure. MLIR [25] is a compiler infrastructure for representing and optimizing code that works across different levels of abstraction, from high-level models like linear algebra arithmetic to low-level hardware instructions. MLIR allows developers to create custom "dialects" tailored to specific domains. A "dialect" defines custom operators and transformations at a specific abstraction level to optimize code for particular hardware architectures. In MLIR, different dialects can be used together in the code representation, enabling collaborative optimizations across various domains to improve overall performance and flexibility.

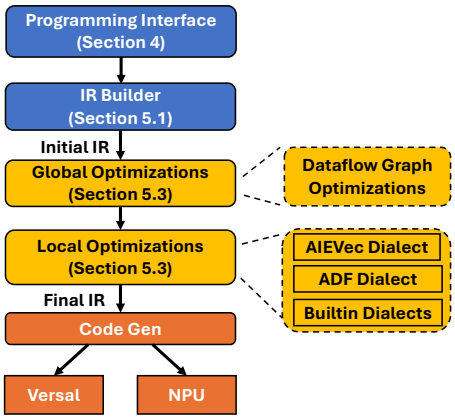


Figure 3: ARIES framework overview.

3 ARIES Framework Overview

While the AIE architectures provide high throughput and energy efficiency, its heterogeneity often poses challenges for users when customizing computation and data communication for general applications. We propose ARIES framework that makes proper abstraction of the parallelism, data movement, and control logic for the AIE-based systems. As shown in Figure 3, ARIES provides users with a novel Python-based programming model for defining customized applications. Then ARIES reuses the open-source Allo [23] compilation flow to generate the initial MLIR assembly. The initial IR will be lowered to the final IR by going through the global and local optimizations proposed by ARIES.

During the global optimizations, ARIES optimizes the high-level dataflow graph without hardware specifics. Then with hardware-related features being specified, ARIES proposes local optimizations for single AIE, AIE array, and PL respectively. For a single AIE kernel, ARIES leverages the existing AIEVec dialect [26] and further introduces transformations to ease the control logic for local buffer locks. For the AIE array, we propose an Adaptive Data Flow (ADF) dialect to describe inter-AIE parallelism and connections. Furthermore, ARIES uses the built-in dialects, e.g., `memref`, `scf`, and `affine`, with additional attributes to represent the customized PL logic. Because both global and local optimizations are under the unified MLIR representation, ARIES enables hardware-agnostic and hardware-specific optimizations to be effectively applied, regardless of the diverse architectures and programming models of various backend devices. We also implement the code generator within MLIR infrastructure translating ARIES final IR to AIE architectures. For Versal, ARIES generates AIE C/C++ intrinsics for AIE cores, C/C++ Vitis ADF APIs for AIE array, HLS C/C++ for PL, configuration file for system connection, and XRT host code for controlling the system. For NPU, ARIES generates the same AIE C/C++ intrinsics for AIE cores, and MLIR-AIE IR/Python binding APIs to manage AIE core connection and host-device data movement.

4 ARIES Programming Model

In this section, we introduce ARIES's Python-based programming model to address the limitations of previous methods. We begin with a motivational example of loop tiling for GEMM kernels, highlighting its challenges. We then present ARIES's programming model, demonstrating how its tile-based interface enhances developer productivity and exposes more opportunities for compiler optimizations, thus providing an efficient framework for exploiting multi-level parallelism on AIE.

4.1 Motivating Example: MLP with Tiled GEMM

We begin with a single GEMM task in the MLP. A basic GEMM task, $C_{ij} = \sum_{k=1}^{K} A_{ik} \cdot B_{kj}$, can be implemented with a three-level loop nest over indices `(i,j,k)`. However, directly mapping this implementation to AIE hardware results in sub-optimal performance. Firstly, the loop dimensions `I`, `J`, and `K` are typically larger than the AIE array's size, preventing them from being fully unrolled and executed in parallel across AIE cores. In fact, improper unroll factors lead to suboptimal performance. Furthermore, memory accesses are inefficient. The inputs and outputs are too large to fit in the on-chip `L1 memory`, so they can only be stored in the slower `L3 memory`; the lack of data locality and on-chip data reuse also leads to decreased memory access efficiency.

```
1  T = int(16)
2  def gemm_tiled(A: T[I, K], B: T[K, J], C: T[I, J]):
3      # Schedule execution order of tiles
4      for i0 in range(0, I, TI_0):
5      for j0 in range(0, J, TJ_0):
6      for k0 in range(0, K, TK_0):
7          DMA_LOAD_L3_TO_L2(...) # On-chip data buffer
8          for i1 in range(0, TI_0, TI_1):
9          for j1 in range(0, TJ_0, TJ_1):
10         for k1 in range(0, TK_0, TK_1):
11             DMA_LOAD_L2_TO_L1(...)
12             # Mapped to cores for parallel execuction
13             for i2 in range(0, TI_1, TI_2):
14             for j2 in range(0, TJ_1, TJ_2):
15             for k2 in range(0, TK_1, TK_2):
16                 LOAD_L1_TO_VECTOR_REGS(...)
17                 for i3 in range(0, TI_2): # SIMD
18                 for j3 in range(0, TJ_2):
19                 for k3 in range(0, TK_2):
20                     MAC(A_vec, B_vec, C_vec)
21                 store(C_vec, C_L1, ...)
22             DMA_STORE_L1_TO_L2(...)
```

Figure 4: GEMM imperative loop tiling in vanilla Python.

To effectively map the GEMM task to AIE, loop tiling is applied. We show a tiled GEMM example written in vanilla Python in Figure 4. It breaks down each original loop axes into multiple nested loops (Lines 4-19) so that these new loop levels can be mapped to different hierarchies in the hardware. The loop axes `(i2,j2,k2)` define the group of inner loop tiles that are distributed across AIE cores for parallel execution. The innermost loop axes `(i3,j3,k3)` are assigned to the vector engine within each AIE core for SIMD processing. The outer loop axes, `(i0,j0,k0)` and `(i1,j1,k1)`, are temporal loops that move through different groups of inner tiles based on the original problem size. As the temporal loops run, frequently accessed data is cached in `L2` and `L1 memory` for faster on-chip access (Lines 7 and 11). This approach allows explicit mapping from loop axis to the hardware architecture, which enables more parallel computation. It also breaks down memory access into smaller chunks, improving on-chip data reuse and data locality.

Loop tiling is crucial for effective mapping to AIE and taking advantage of multi-level parallelism. However, the imperative loop tiling approach in Figure 4 creates new challenges. First, it requires significant effort in code restructuring; users need to create deep nested loop structures, decide the mapping from loop levels to AIE cores, and coordinate data movement between memory hierarchies explicitly. Second, data reuse between loop tiles is implicit; it often requires additional compiler analysis like polyhedral modeling [27] to identify reusable data across the tiled loops and avoid repeated off-chip memory access. This makes it harder to optimize on-chip data reuse when mapping loop tiles to parallel compute units.

4.2 Tile-Based Task Abstraction

To address these challenges, ARIES introduces a tile-based task abstraction that simplifies loop tiling by hiding aforementioned complexities from programmers. Figure 5 shows a code example of a tiled GEMM task in ARIES. The function computes a small tile of the overall problem, with tiling applied to the `i`, `j`, and `k` dimensions. This breaks the original problem size into a 3D grid of smaller tiles. When all the tiles in the grid are processed, the entire task is complete.

```
1  @aries.task_tile
2  def gemm(A: T[I, K], B: T[K, J], C: T[I, J]):
3      i, j, k = aries.tile_ranks()    # Tile ranks
4      TI, TJ, TK = aries.tile_sizes() # Tile sizes
5      ti = arange(i*TI, (i+1)*TI)  # I tile range
6      tj = arange(j*TJ, (j+1)*TJ)  # J tile range
7      tk = arange(k*TK, (k+1)*TK)  # K tile range
8      A_L1: T[TI, TK] = A[ti, tk]  # Load to L1 in core
9      B_L1: T[TK, TJ] = B[tk, tj]  # Load to L1 in core
10     C_L1: T[TI, TJ] = C[ti, tj]  # Load to L1 in core
11     for i in range(0, TI):
12       for j in range(0, TJ):
13         for k in range(0, TK):
14           C_L1[i, j] += A_L1[i, k] * B_L1[k, j]
15     C[ti, tj] = C_L1             # Store partial sum
```

Figure 5: GEMM task with tile abstraction in ARIES — tile ranks (Line 5) refers to the index number of tile in each dimension

Grid Semantics for Tiles. Inside the task tile, programmers can access tile indices in the grid (Line 3), and define the computation to be performed by each tile using its ranks and external memory pointers. The tile size for each dimension (Line 4) can be adjusted outside the function body based on the requirement. The semantics to index a tile in the grid (Lines 3-4) is borrowed from CUDA [28] and Triton [22]. Compared with the imperative loop tiling approach, it makes the code more concise without compromising expressiveness. A special case is that when a task has only one tile in its grid. In this situation, the tile size matches the original problem size of `(I,J,K)`, and no tile-level parallelism is exploited within the task.

Tile-Based Memory Abstraction. In a task tile, all its memories are described at the granularity of a tile. The input parameters representing external memories of a tile outside the AIE core (Line

2) is mapped to `L2` or `L3 memory`, and the local memory is mapped to the `L1 memory` inside an AIE core (Lines 8-10). This tile-based abstraction clearly exposes opportunities for data reuse between task tiles by representing memory access as hyper-rectangles in the memory space. By analyzing the overlap between the memory regions of task tiles, the compiler can determine how to reuse input and output data on-chip. For example, task tiles along the `i` dimension in the grid share the same memory tile of `B` matrix, so these memory tiles can be cached in `L2 memory` and broadcast to the task tiles as needed.

Tile Scheduling for Multi-Level Parallelism. Each task tile is designed to run on a single AIE core. However, when the number of task tiles exceeds the available AIE cores, or tiles from different tasks need to share these cores, a scheduling strategy is required to distribute them efficiently to exploit task- and tile-level parallelism. Additionally, within each core, scheduling techniques like pipelining and SIMD are needed to maximize intra-core parallelism. To address these requirements, ARIES adopts the concept of decoupled scheduling from Allo [23] and provides a set of scheduling primitives as shown in Table 1. These allow users to customize how task tiles are mapped across AIE cores and executed on the AIE cores without modifying the algorithm defined in the task tile function.

Table 1: Tile scheduling primitives in ARIES – User-level control over multi-level parallelism both across and within AIE cores.

Primitive	Description
`.to(tiles, cores)`	Map tile(s) of a task to designated AIE core(s). ❶ Task-level and ❷ Tile-level parallelism
`.pipeline(axis, factor)`	Enable instruction pipelining at loop axis. ❸ Loop-level parallelism
`.vectorize(axis, factor)`	Apply vectorization over loops with factors. ❹ Data-level parallelism

ARIES introduces (1) `.to()` primitive allows users to map tiles of different tasks across AIE cores to exploit task- and tile-level parallelism. If multiple task tiles are mapped to a single AIE core, hidden temporal loop levels are created to schedule and execute the tiles sequentially on that core. (2) The `.pipeline()` primitive enables loop pipelining on a task tile to exploit loop-level parallelism on a single AIE core. (3) The `.vectorize()` primitive applies vectorized processing to certain loop axes of a task tile, leveraging data-level parallelism in the AIE core. We provide code examples in next subsection to demonstrate the usage of these primitives.

4.3 Task- and Tile-Level Parallelism

In this section, we demonstrate how the tiles of each task are scheduled to AIE cores for parallel execution. We show the code example of a two-layer MLP in Figure 6. In this example, we use an AMD Ryzen-AI NPU device with a 4×5 array of AIE cores as the hardware target. The tile size for each task can be customized using the subscript operator (Lines 3-4). Each task returns a handle, which is used for scheduling the task on AIE cores. A schedule object is created to apply the customization for `task0` and `task1` (Line 7).

Tile-to-Core Mapping. We use the `.to()` primitive to assign tiles from two tasks to separate AIE core groups for parallel execution. ARIES offers two ways to map task tiles to AIE cores: programmers

```
1  # Configure task's tile sizes and task dependency
2  grid, size = (I/TI, J/TJ, K/TK), (TI, TJ, TK)
3  task0 = gemm[grid, size](A, W0, B) # B = A @ W0
4  task1 = gemm[grid, size](B, W1, C) # C = B @ W1
5
6  from aries.targets import NPU
7  sch = aries.Schedule([task0, task1])
8  # Case 1: Automatic tiles scheduling
9  sch.to(task0.tiles(), NPU[:4, :2])
10 # Case 2: Explicit tile mapping to AIE cores
11 for (i, j, k), tile in enumerate(task1.tiles()):
12     sch.to(tile, NPU[i%4, 2+j%2])
```

Figure 6: Mapping task and its tiles to AIE cores.

can either map all task tiles to a core group (Line 9) or explicitly assign each task tile to specific cores (Line 12). In the first case, ARIES uses an adaptive core placement algorithm to determine tile placement that can minimize data movement cost, explained in Section 5.3. For the second GEMM task in our example, its task tiles are explicitly mapped along the grid axes `(i, j, ...)` to AIE cores `NPU[:4, 2:4]`. This causes the task tiles along the `k` axis to be computed on the same AIE core in sequential order, and the partial sums over `k` dimension remain on the same core. As a result, the GEMM is computed in output-stationary dataflow fashion [29].

Cross-Tile Communication. Data movement between AIE cores is automatically determined by the task tile placement set by the user. When task tiles are on adjacent AIE cores, they can exchange data through the fast DMA interface, where one core directly accesses the `L1 memory` of another. When the communicating task tiles are on non-adjacent AIE cores, data can be directly transferred via the AXIS streaming interface through hops of on-chip switches. If a core requires inputs from multiple other cores but exceeds its input stream channel limit, the data is first gathered in `L2/L3 memory` before being sent to the destination core for processing.

4.4 Intra-Tile Parallelism

Continuing with the tiled MLP example, we demonstrate how to optimize the task tiles mapped to each AIE core to maximize parallelism within the core. The code example is shown in Figure 7.

```
1 axes = task0.get_loops()
2 # Loop-level parallelism: instruction pipeline
3 sch.pipeline(axes[0], range=(1, 64))
4 # Data-level parallelism: SIMD vectorization
5 sch.vectorize(axes, factors=[4, 8, 4])
```

Figure 7: Intra-core parallelism for each tile.

Loop-Level Parallelism. Users can use `.pipeline()` primitive to enable instruction pipelining for the target loops within a task tile (Line 3). This primitive inserts a directive into the low-level generated code, allowing the single-AIE-core compiler to schedule loop operations in a pipelined manner. The `factor` option specifies the desired initiation interval as a hint to guide the low-level vendor compiler in scheduling instructions.

Data-Level Parallelism. The `.vectorize()` primitive maps specified loop axes within a task tile into a format that can be processed by SIMD instructions on a single AIE core. Specifically, this involves an additional level of loop tiling based on the `factors` provided by the user. The memory tiles within the loop are loaded into dedicated vector registers, and the generated low-level code will use SIMD-specific instructions to perform operations in parallel.

MLIR program generated by ARIES IR builder

```
func.func @top_gemm(%A, %B, %C){
 affine.for %i0 = 0 to I step TI_0 {          Ⓐ L3(T)
 affine.for %j0 = 0 to J step TJ_0 {          off-chip
 affine.for %k0 = 0 to K step TK_0 {
  affine.for %i1 = 0 to TI_0 step TI_1 {
  affine.for %j1 = 0 to TJ_0 step TJ_1 {      Ⓑ L2(T)
  affine.for %k1 = 0 to TK_0 step TK_1 {      on-chip
   adf.cell @cell0 {
    affine.for %i2 = 0 to TI_1 step TI_2 {
    affine.for %j2 = 0 to TJ_1 step TJ_2 {    Ⓒ L1(S)
ⓐ affine.for %k2 = 0 to TK_1 step TK_2 {      PE array
      %off0 = %i2 + %i1 + %i0
      %off1 = %j2 + %j1 + %j0
      %off2 = %k2 + %k1 + %k0
  ⓑ adf.dma(%A[%off0,%off2][TI_2,TJ_2][1,1], %A_loc)
      adf.dma(%B[%off2,%off1][TJ_2,TK_2][1,1], %B_loc)
      adf.dma(%C[%off0,%off1][TI_2,TK_2][1,1], %C_loc)
      func.call @gemm(%A_loc, %B_loc, %C_loc)
      adf.dma(%C_loc, %C[%off0, %off1][TI_2,TK_2][1,1]
   }}}}
}}}}}}}

func.func @gemm(%A_loc, %B_loc, %C_loc){
 affine.for %i3 = 0 to TI_2 {                 Ⓓ L1(T)
  affine.for %j3 = 0 to TJ_2 {                single core
   affine.for %k3 = 0 to TK_2 {
     %0 = affine.load %A_loc[%i3, %k3]
     %1 = affine.load %B_loc[%k3, %j3]
     %2 = arith.muli %0, %1 : i32
     %3 = affine.load %C_loc[%i3, %j3]
     %4 = arith.addi %3, %2 : i32
     affine.store %4, %C_loc[%i3, %j3]
}}}}
```

Figure 8: ARIES initial IR including a single AIE, AIE array, PL, and top configuration of a GEMM kernel.

5 ARIES Representation and Optimizations

We further introduce the IR underlying the proposed programming models in Section 5.1 and 5.2. We provide a detailed description of the ARIES initial IR generated by the IR builder and the final IR fed into the code generation. We then elaborate on the ARIES optimizations and automation in Section 5.3 that transform the initial IR to the final IR, delivering significant performance improvement.

5.1 ARIES Representation Overview

ARIES IR Builder and Initial IR. ARIES leverages Allo [23] as the IR builder to translate our frontend program to MLIR representation. The GEMM illustrated in Figure 5 is translated into the IR shown in Figure 8 with the same loop structure marked by 8Ⓐ-8Ⓓ. The space-time transformation is presented in the function **top_gemm** where loop bands 8Ⓐ, 8Ⓑ, 8Ⓒ represent L3, L2 temporal mapping and L1 spatial mapping respectively. To identify the spatial loops, the `adf.cell` operation is created so that the `For` loops within its region can be recognized as spatial mapping during the later stage. The `adf.cell` supports up to one reduction loop (8ⓐ) and two non-reduction loops constructing a 3D array that better explores the IO reuse opportunity. The L1 temporal loop band 8Ⓓ that describes the computation of an AIE core is extracted in the function **gemm** and called within function **top_gemm**. ARIES IR builder will extract the data movement to/from the AIE using the `adf.dma` operations. The `adf.dma` is a high-level abstraction for n-dimensional data slicing. It moves data between two memory values, for example in 8ⓑ, from external memory `%A` (source) to the L1 local memory `%A_loc` (destination) with three pairs of data slicing information [offset0, offset1], [size0, size1], and [stride0, stride1].

ARIES Final IR before Translation. ARIES embraces the MLIR ecosystem, enabling the seamless composition of different dialects to describe our targeted heterogeneous system, including the AIE array, AIE core, and PL. The ARIES final IR of a GEMM example is shown in Figure 9. Overall, the GEMM is spatially mapped to two AIEs through the reduction dimension K, and can be expressed as A0×B0 + A1×B1 = C. The conceptual graph of the IR is presented in snippet ❶ where the data is initially stored in off-chip `L3 memory`. The AXI4 stream loads the data into `L2 memory` to enhance data reuse. It is then transferred to the AIE array via the AXIS/PLIOs. ARIES explores the inter-AIE data forwarding optimization that will be introduced in Section 5.3. Instead of evicting data out of AIE `L1 memory` to `L2 memory`, it allows the output temporary data to be transferred to other spatial AIEs through either the AXIS connection or shared memory. The code snippet ❷ serves as the top function that instantiated the AIE array graph (9ⓑ) called `adf_cell0` and the PL function (9ⓒ) called `func_pl`. It also defines the top-level connections in 9ⓐ between the data streams of PL and the ports in the AIE shim interface tile. In code snippets ❸, ❹, and ❺, ARIES utilizes the proposed ADF dialect, the existing AIEVec, and MLIR built-in dialects to represent AIE array, single AIE, and PL programs, respectively. These snippets will be further explained in the following section.

5.2 ARIES Final IRs

ADF Dialect for AIE Array Construction. We propose an ADF dialect for exploring inter-tile level parallelism. As illustrated in snippet ❸, it exposes the IOs to the outside of the AIE array, allocates multiple AIE cores, and defines the connection among AIEs. More specifically, in the function `adf_cell0` at 9ⓓ, it defines the input and output PLIOs of the graph which corresponds to the 5 AXIS to AIE connections shown in snippet ❶. ARIES exposes the IO width and placement configuration through the `adf.config` operation. 9ⓔ sets `%plioA0` to 128 bits and places it at interface tile [30,4]. The `adf.connect` is used to establish the connection between the port to AIE local memory as well as the connection among AIE local memories. For example, while 9ⓕ connects PLIO `%plioA0` to the local memory of AIE 0, 9ⓗ performs an L1 data forwarding between local memory of AIE0 and AIE1. The corresponding conceptual connection is also demonstrated in snippet ❶. At last `func.call` (9ⓖ) is used to allocate the AIEs with the core placement marked by [col, row].

AIEVec Dialect for Single AIE Core. ARIES reuses the existing AIEVec dialect proposed in MLIR-AIE [11] to represent the computation in an AIE core. The AIEVec dialect creates a precise MLIR representation for the AIE intrinsics [30]. By using this dialect, ARIES provides a vector version of the intra-AIE computation shown in snippet ❹. At 9ⓘ and 9ⓙ, it packs the data in %j3 dimension by 8 and uses the vector operations to explore the data-level (SIMD) and instruction-level (VLIW) parallelism of AIE [31].

Built-in Dialects and Directives for PL. The functions of PL are presented by the builtin-dialects including `memref`, `scf`, `affine`, etc. in snippet ❺. During the optimization stage, HLS-related directives including *pipeline*, *dataflow*, *inline*, *bind_storage*, and *interface* will be automatically inferred to improve the performance as shown in 9Ⓛ. In the GEMM example on Versal, the on-chip BRAMs/URAMs are allocated in the PL to enhance data reuse, thereby preventing computation from being bounded by the off-chip memory access. As marked by 9ⓚ, the PL function acts as the data mover,

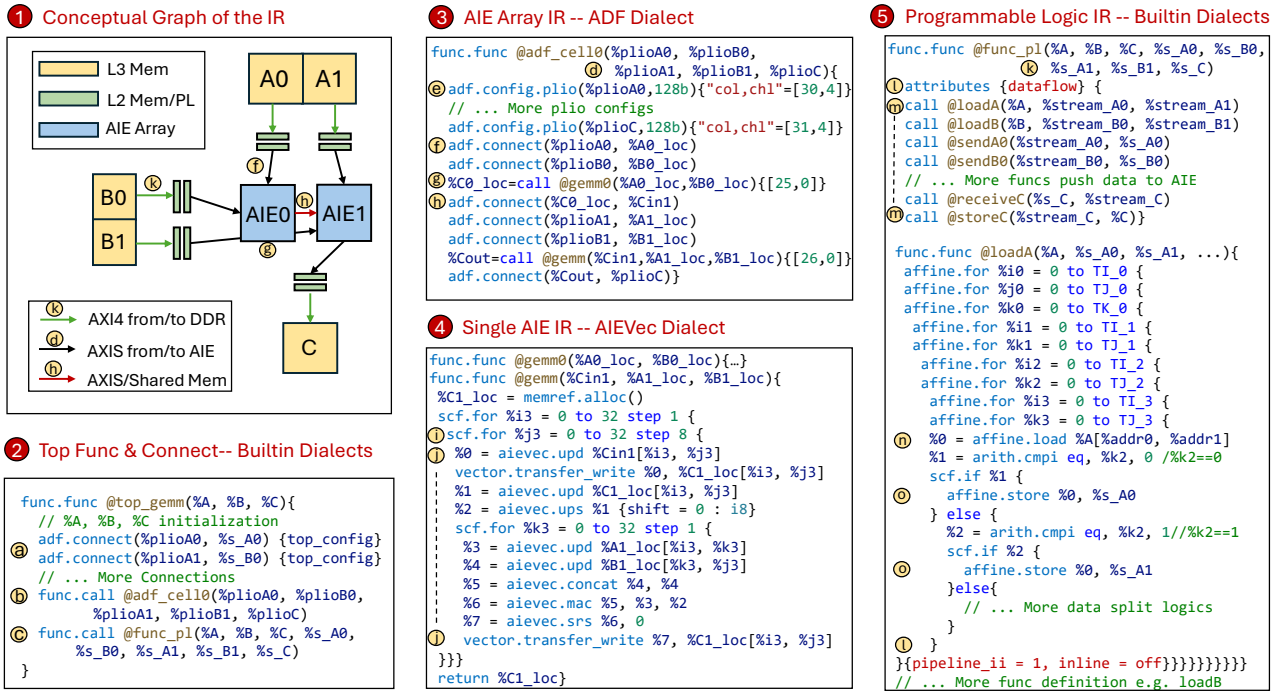


Figure 9: ARIES representation including single AIE, AIE array, PL, and top configuration of a GEMM example.

Table 2: Summerization of passes in ARIES framework.

Opt Level	Objects	Passes	Descriptions
Global	Dataflow Graph	-broadcast-detection	Detect DMA broadcast
		-data-forwarding	Enable L1 data forwarding
Local	AIE Array	-dma-to-io	Convert memory dma by IOs and connects
		-adf-cell-create	Create the ADF graph connections
		-io-materialize[1]	Materialize IO to GMIO or PLIO
		-io-packing[1]	Specify the AIE IO width
		-core-placement[1]	Place 3D logic array on 2D physical layout
		-io-placement[1]	Place IOs on the AIE IO tiles
	AIE Core	-aie-vectorize[2]	Vectorize specific loops by factors
		-kernel-interface	Enable automatic lock control for cores
	PL	-axi-packing	Increase the AXI width for higher DDR BW
		-burst-detection	Increase DDR burst by merging interleaved access
		-pl-double-buffer	Create double buffers for the functions in PL
		-pipeline-ii[3]	Specify the pipeline II of functions in PL

Note: [1] with parameter from primitive `.to()`; [2] with parameter from primitive `.vectorize()`; [3] with parameter from primitive `.pipeline()`.

transferring data from off-chip memory (%A, %B, %C) to the on-chip buffer and sending it to the streams (`%s_xxx`) that are finally connected to the AIE array for computation. The sub-functions are defined in the top function of PL adopting a full task-level pipeline in HLS design under *dataflow* pragma (9ⓜ). For illustration purposes, we provide the function definition of `LoadA`, where the data is read from the off-chip memory `%A` to the register `%1` using `affine.load` operation (9ⓝ) and is interleaved written to the corresponding internal stream by `affine.store`(9ⓞ) respectively.

5.3 ARIES Optimizations and Automation

ARIES introduces a set of automated optimization and transformation passes that lower the initial IR to the final IR. These optimizations are categorized into global, which operate at the graph level, and local, which target code sections mapped to the specific hardware hierarchy, as summarized in Table 2.

Global Optimizations. When conducting global optimizations, ARIES decouples the high-level dataflow graph optimizations from

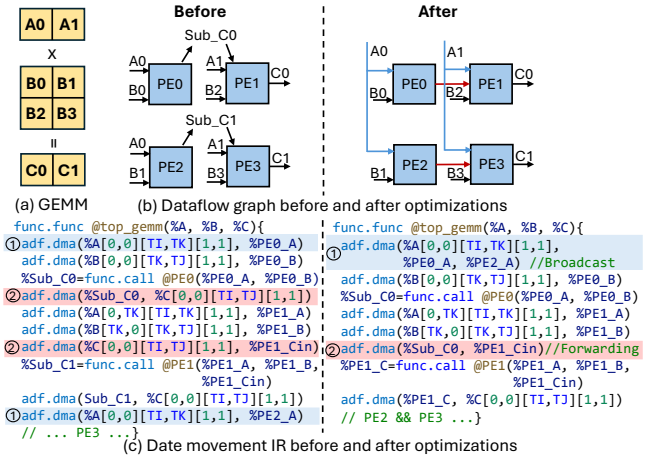
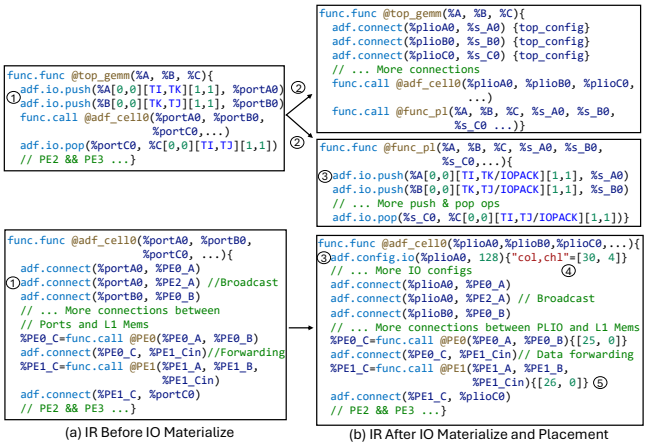


Figure 10: ARIES global optimizations.

the hardware specifications. It allows for flexible and scalable optimizations that can be refined for specific hardware components or architectures in a later stage. We use another GEMM as an example shown in Figure 10(a) to demonstrate the global passes including `broadcast-detection` and `data-forwarding`. As introduced in Section 5.1, ARIES IR builder will extract the data movement where the conceptual graph and its IR are shown in the `Before` side of Figure 10(b) and (c). The tiles corresponding to each PE, e.g., `A0`, are sent to the local buffer of the PE (`%PE0_A`) through the `adf.dma` operation. Before conducting the global passes, the same tile will be loaded from the external memory to the local memory repetitively as marked by 10①. Besides, the intermediate result from `PE0` and `PE2` will be stored to the external memory and loaded back as marked by 10②. The conceptual graph and the IR after optimization are also demonstrated in Figure 10. To avoid loading



(a) IR Before IO Materialize (b) IR After IO Materialize and Placement

Figure 11: ARIES local optimizations.

the same tile of data multiple times, the `broadcast-detection` pass will walk through all the `adf.dma` operations to gather the destination memories of the ones with the same source memories. The `data-forwarding` pass will optimize the data movement by eliminating the consecutive store and load pairs that access the same tile of external memory.

Local Optimizations. After the global optimizations, the passes for local optimizations will be called following the order listed in Table 2. The `adf.dma` operations will be converted to `adf.io.push` or `adf.io.pop` operations to involve the IO information using the `dma-to-io` pass. Instead of moving data between two memories, the data will be pushed or popped from one memory to an IO port and then the IO port will be connected to the other memory as marked by 11①. The `adf-cell-create` pass then extracts the construction of the AIE array into the function `adf_cell0` shown in Figure 11(a). According to the user-specified IO type from `.to()` primitive in the programming interface, the IO port will be materialized to GMIO or PLIO by `io-materialize` pass where GMIO and PLIO refer to the IOs connecting the AIE array with external memory and PL respectively. For designs with PLIO, a function representing the logic in the PL side will be created through this pass(11②). In order to balance the IO bandwidth to the AIE array, the `io-packing` pass will configure the materialized IO to the specified port width, frequency, and burst length(11③). The size and data type of the memory in `adf.io.push` and `adf.io.pop` operations will also be adjusted (11③).

ARIES provides the default AIE core placement algorithms and the IO placement algorithm to help alleviate routing congestion when scaling to hundreds of AIEs. The AIE core placement algorithm aims to map the 3D logic AIE array to the 2D physical AIE array. To leverage the shared memory and cascade IO connections of the AIEs in the reduction dimension (dim), the cores are required to be placed in adjacent locations. ARIES designs three placers to allocate the AIEs in reduction dims horizontally-adjacent, vertically-adjection, and zigzag-adjacent. We demonstrate the horizontal placer in Algorithm 1. For each core, it takes as inputs the 3D AIE ID (`i, j, k`), the total number of AIEs in each dim (`PI, PJ, PK`), the column and row offset (`colOff, rowOff`), and the available AIEs in the physical 2D array (`colNum, rowNum`). Based

Algorithm 1 AIE Core Placement Algorithm

Input: i, j, k, PI, PJ, PK, colOff, rolOff, colNum, rowNum
Output: col_pos, row_pos

```
1:  function HORIZONTAL_PLACER(i, j, k, PI, PJ, PK, colNum, rowNum)
2:      curRowNum = rowNum - rolOff
3:      if PJ >= PI then
4:          height = min(PJ, curRowNum)
5:          serialized_ij_id = j + i * PJ
6:      else
7:          height = min(PI, curRowNum)
8:          serialized_ij_id = i + j * PJ
9:      colReq = ceil( PI * PJ * PK / height)
10:     if colReq > colNum then
11:         return false;
12:     remi = serialized_ij_id % height
13:     quot = serialized_ij_id // height
14:     pid = remi + PK * height + quot * PK * height
15:     return pid, height
16: function MAIN_PLACER(i, j, k, PI, PJ, PK, colOff, rolOff, colNum, rowNum)
17:     pid, height = HORIZENTAL_PLACER(i, j, k, PI, PJ, PK, colNum, rowNum)
18:     col_pos = colOff + pid // height
19:     row_pos = rowOff + pid % height
```

on the placement ID (`pid`) and `height` generated by different placers, the column and row position (`col_pos, row_pos`) of an AIE core can be calculated by Lines 18-19. Without loss of generality, we assume `k` is the reduction dim in the algorithm. For the horizontal placer, the AIEs cores with the same ID in dim `i` and `j`, and consecutive ID in dim `k` will be placed in the same row but in different columns. Then by judging the total number of AIEs in dim `i` and `j`, it calculates the serialized `i` and `j` IDs (`serialized_ij_id`) which determines the AIE cores to be placed in adjacent rows (Lines 3-8). The `height` of the physical 2D AIE array is chosen from the minimum value of PI/PJ and the number of physical rows in the AIE array. In Lines 12-14, the pid can be generated based on the `serialized_ij_id` and `height`. ARIES offers the user the extensibility of defining customized placement algorithms using `.to()` primitives introduced in Section 4.3. Based on the core placement algorithm, ARIES applies a routing-aware IO placement algorithm proposed in [32] for both PLIO and GMIO where it places the IOs in the available channel of the interface tile that leads to the least west and east routing congestion. By running the `core-placement` and `io-placement` passes, the position of each IO and core will be attached as marked by 11④ and 11⑤. For AIE core optimizations, based on the user-specified configurations, ARIES applies the vectorization over loops using the affine passes including `affine-unroll` and `affine-super-vectorize`. Then `aie-vectorize` pass in MLIR-AIE [11] is utilized to generate the AIE vector IR in `aievec` dialect. To make the current flow compatible for both Vitis ADF and pure AIE compilers, `kernel-interface` pass is used to modify the type of AIE kernel interfaces.

On the PL side, the `adf.io.push` and `adf.io.pop` marked by 11③ will further be lowered to the builtin `memref`, `scf`, and `affine` dialects before code generation. ARIES implements passes to make full use of the off-chip bandwidth by automatically packing AXI4 port width (`axi-packing`) and detecting opportunities to increase the burst length(`burst-detection`). The double buffer technique is utilized by the pass `pl-double-buffer` to overlap the time spent on off-chip access with the on-chip processing time. ARIES allows users to customize their designs with the `pipeline_ii` configuration, defaulting to 1 if not specified.

Overall Optimization Insights. Adopting the MLIR ecosystem, ARIES optimizes designs globally and locally, ensuring extensibility and reusability for future AIE architectures. For global optimizations, it abstracts dataflow without hardware specifics (e.g., PLIO, GMIO, Mem-tile), allowing for the analysis of general data movement optimizations, such as broadcast and packet-switch patterns, as well as top-down customization of dataflows like output-stationary dataflow and systolic-array architectures. Unlike bottom-up approaches, which require significant modifications across components when optimizing one (e.g., AIE array, core, or Mem-tile), this top-down method minimizes the changes in the other components. For local optimizations, all components remain a unified IR, ensuring that local optimizations are aware of the other components. For example, adjusting the IO port width in the AIE array will automatically trigger updates to memory data types for the buffer shown in 11③. ARIES offers a unified, open-source framework that provides optimized solutions for different AIE-related backends, while also enabling users to explore customized optimizations.

6 Evaluation

This section presents on-board evaluation results of ARIES using realistic benchmarks. For Versal ACAP AIE architectures, we use the VCK190 evaluation board, featuring 8×50 AIE cores, 90K LUTs, 1968 DSP58s, 967 BRAMs, 463 URAMs, and one 25.6GB/s DDR4-DIMM. For all the experiments on Versal, ARIES uses AMD Vitis version 2023.1 as our backend compiler. To measure power consumption, each benchmark is executed for over 60 seconds, and the average power is reported using the AMD board evaluation and management tool [33]. For Ryzen-AI NPU architectures, the tests are conducted on an AMD Ryzen™ 9 7940HS CPU, which includes an integrated NPU with 4×5 AIE-ML cores. We use AMD Vitis 2023.2 and MLIR-AIE [11] for NPU compilation.

6.1 Single-Kernel Benchmarks

We first evaluate widely adopted tensor algebra benchmarks used in [6, 15, 7, 34, 35, 36]. The algorithms of the benchmarks including GEMM, TTM, TTMc, and MTTKRP are defined by equation (1)-(4).

$$GEMM: C(i,j) += A(i,k) \times B(k,j) \quad (1)$$

$$TTM: C(i,j,k) += A(i,j,l) \times B(l,k) \quad (2)$$

$$TTMc: D(i,j,k) += A(i,l,m) \times B(l,j) \times C(m,k) \quad (3)$$

$$MTTKRP: D(i,j) += A(i,k,l) \times B(k,j) \times C(l,j) \quad (4)$$

GEMM Benchmarks under Various Data Types. For the GEMM benchmark, we compare the on-board throughput and energy efficiency on Versal VCK190 with the SOTA solutions [15, 7] under FP32, INT16, and INT8 data types in Table 3. The I * J * K represents the AIE array parallelism factors on the three dimensions of GEMM. For the intra-AIE design, all the GEMM benchmarks apply the well-optimized kernel proposed by CHARM23 [6]. CHARM24 [7] and AutoMM [15] extensively explore the PLIO reuse by applying combined broadcast and packet-switch connections. Although it requires less number of PLIOs, their data reuse pattern relies greatly on the computation-to-communication ratio of the single core. ARIES explores broadcast opportunities, proposes different core placement algorithms, and applies a routing-aware IO placement algorithm, thus achieving 1.2× and 1.7× higher AIE utilization under INT16 and INT8 data types without performance degradation. CHARM24 [7] and AutoMM [15] improve the AIE array efficiency by adopting fine-grained cascade connection between AIEs and the bubble-free data movement between PL and AIEs. However, due to the double buffer synchronization HLS style on the PL side, the communication efficiency for the AIE array is around 80%. In contrast, ARIES adopts a full task-level pipeline HLS style which achieves around 95% AIE array efficiency. The analysis can also be verified through our experiments. With FP32 data, ARIES uses 352 AIEs – 8% fewer than two baseline designs – while achieving 1.31× and 1.18× higher throughput compared to CHARM24 [7] and AutoMM [15]. As these three designs share the same intra-AIE optimization, ARIES achieves the gain on throughput from the efficient data movement between PL and AIE array. For INT16 and INT8 data types with higher AIE utilization and AIE array efficiency, ARIES achieves up to 2.11× and 1.63× throughput gain compared to AutoMM [15]. The higher throughput also leads to higher power consumption in ARIES designs. However, ARIES still achieves up to 1.20×, 1.57×, and 1.35× better energy efficiency compared to AutoMM [15] for FP32, INT16 and INT8 data types.

TTM, TTMc, and MTTKRP Benchmarks. For TTM, TTMc, and MTTKRP benchmarks, we conduct experiments on VCK190 under INT32 data type. The tiling factors, resource utilization, and throughput are summarized in Table 4. We tile each loop 4 times where the loop index from 0-3 represents the memory hierarchy from external to internal. More specifically, indices i3-m3 represent the data stored in AIE local memory. Indices i2-m2 are the spatial loops unrolled on the AIE array. Indices i1-m1 are the array partitioning loops that determine the data reuse on the SRAM of the PL side. Indices i0-m0 represent the off-chip to on-chip data movement. Based on our core and IO placement algorithms, we utilize 352, 192, and 192 AIEs for TTM, TTMc, and MTTKRP benchmarks. ARIES achieves 4.9, 4.8, and 4.8 TOPS under INT32 data type with 87%, 80.6%, 80.6% overall AIE efficiency. Note that the throughput for TTMc and MTTKRP is an effective throughput where we explore algorithm optimization to reduce the multiply operations.

Benefits of ARIES Compilation Framework. To demonstrate the advantages of ARIES in enhancing productivity and improving the success rate of AIE-related designs, we present an evaluation in Table 5. It compares lines of code (LoC), ARIES compilation time, AIE placement and routing (PnR) time, and PnR results across GEMM benchmarks with AIEs ranging from tens to hundreds. Versal VCK190 and AMD Vitis were selected as the backend device and tool to illustrate the benefits of ARIES. The compilation is running on a virtual machine of an Intel Xeon Gold 6346 CPU with 32 threads and 128GB memory enabled. Using ARIES Python-based programming model, only 25 LoC is required from the user for all design cases with small modifications to the number of grids. We also report the ARIES compilation time, Vitis AIE PnR time, and the corresponding PnR result when the AIE core and IO placement optimizations proposed by ARIES are enabled and disabled. The ARIES compilation time is from the initial IR to the final code after the generator. The Vitis AIE PnR time is from the vendor tool report for compiling the generated design. By applying our AIE core and IO placement optimizations, ARIES reduces the average PnR time from 2300s to 64.9s with 0.3s of additional compilation time. For designs with over 128 AIEs, ARIES core and IO placement algorithm enables successful placement and routing, whereas no solution can be found without ARIES optimizations.

Table 3: On-board throughput and power comparisons of GEMM benchmark under FP32, INT16, INT8 data types.

DType	Works	Para (I*J*K)	LUT	BRAM	URAM	DSP	PLIOs	AIE	TOPS	Power(W)	GOPS/W
FP32	**ARIES (Ours)**	**11x8x4**	**191,324(21.26%)**	**855 (88.42%)**	**352 (76.03%)**	**374 (19.00%)**	**76 + 88**	**352 (88%)**	**4.92 (1.31x)**	**63.8**	**77.1 (1.20x)**
	CHARM [7]	12x8x4	103,959(11.55%)	764 (79.01%)	384 (82.94%)	165 (8.38%)	72 + 24	384 (96%)	4.18 (1.11x)	61.9	67.5 (1.05x)
	AutoMM [15]	12x8x4	64,849(7.20%)	661 (68.36%)	384 (82.94%)	163 (8.28%)	72 + 24	384 (96%)	3.75 (1.00x)	58.3	64.3 (1.00x)
INT16	**ARIES (Ours)**	**11x8x4**	**184,373(20.49%)**	**631 (65.25%)**	**352 (76.03%)**	**46 (2.34%)**	**76 + 88**	**352 (88%)**	**15.86 (2.11x)**	**76.3**	**207.9 (1.57x)**
	CHARM [7]	12x3x8	111,626(12.41%)	885 (91.52%)	384 (82.94%)	91 (4.62%)	72 + 48	288 (72%)	10.03 (1.34x)	64.8	154.8 (1.17x)
	AutoMM [15]	12x3x8	92663(10.30%)	477 (49.33%)	384 (82.94%)	93 (4.73%)	72 + 48	288 (72%)	7.51 (1.00x)	56.8	132.2 (1.00x)
INT8	**ARIES (Ours)**	**10x8x4**	**144,825(16.09%)**	**823 (85.11%)**	**320 (69.11%)**	**0 (0.00%)**	**72 + 80**	**320 (80%)**	**45.94 (1.63x)**	**73.8**	**622.5 (1.35x)**
	CHARM [7]	8x6x4	115,628(12.85%)	662 (68.46%)	388 (83.80%)	71 (3.61%)	80 + 24	192 (48%)	31.31 (1.11x)	62.7	499.4 (1.08x)
	AutoMM [15]	8x6x4	85,073(9.45%)	669 (69.18%)	384 (82.94%)	71 (3.61%)	80 + 24	192 (48%)	28.15 (1.00x)	61.0	461.5 (1.00x)

Table 4: Tiling factors, resource utilization and throughput of TTM, TTMc and MTTKRP under INT32 data type.

	TTM	TTMc	MTTKRP
i0,j0,k0,l0,m0	2,2,2,64,-	2,2,2,8,8	2,2,8,8,-
i1,j1,k1,l1,m1	1,4,6,1,-	1,2,2,8,8	2,8,6,4,-
i2,j2,k2,l2,m2	1,11,8,4,-	1,8,12,1,2	8,12,1,2,-
i3,j3,k3,l3,m3	1,32,32,32,-	2,16,16,16,32	2,32,16,32,-
LUT	198510(22.06%)	142266(15.81%)	192936(21.44%)
BRAM	855(88.42%)	286.5(29.63%)	942.5(97.47%)
URAM	352(76.03%)	128(27.65%)	448(96.76%)
DSP	20(1.02%)	0(0.00%)	0(0.00%)
PLIOs	76+88	34+96	52+96
AIEs	352(88%)	192(48%)	192(48%)
TOPS	**4.9**	**4.8**	**4.8**
AIE Efficiency	**87%**	**80.6%**	**80.6%**

Table 5: Lines of code (LoC), ARIES and AIE PnR compilation time comparison across different AIE scales for GEMM — CIP refers to AIE core and IO placement optimizations in ARIES.

AIEs	ARIES LoC	CIP Enabled	ARIES Comp.	AIE PnR Time	PnR
32	~25	N	0.26s	19.7s	✓
		Y	**0.28s**	**10.4s**	✓
64	~25	N	0.42s	1828.2s	✓
		Y	**0.45s**	**23.7s**	✓
128	~25	N	0.89s	1874.7s	✓
		Y	**0.96s**	**44.1s**	✓
256	~25	N	2.29s	>3600s	×
		Y	**2.47s**	**110.9s**	✓
320	~25	N	3.28s	>4200s	×
		Y	**3.49s**	**135.3s**	✓

6.2 Multi-Layer Applications

Residual Neural Network Layer. We implement a residual neural network layer from ResNet [37] in ARIES. Our evaluation focuses on the performance of different design configurations by toggling scheduling primitives in ARIES when mapping the target application to the NPU device. The ResNet layer processes an input image of size 64x64 with 256 channels in the INT8 data type. It comprises three tasks: (1) a Conv1×1 task with 256 input channels (IC) and 64 output channels (OC), (2) a Conv3×3 task with 64 IC and 64 OC, and (3) another Conv1×1 task with 64 IC and 256 OC, fused with an element-wise addition operator for the skip connection. We conducted an ablation study using various scheduling primitives in ARIES and compared the results to the optimized INT8 NPU design from Ryzen-AI-SW [20]. The results are presented in Table 6.

In design D1 of ARIES, these tasks are mapped to three adjacent AIE cores respectively to take advantage of task-level parallelism.

Table 6: ResNet layer evaluation on Ryzen-AI NPU — SL stands for SIMD lane; IP indicates if instruction pipeline is enabled. Util means the utilization of all AIE cores on the Ryzen-AI NPU device. RT refers to run time excluding host-side memory copy overhead.

Designs	Tile	SL	IP	Util (%)	RT (ms)	Speedup
D1 scalar	[16, 16]	1	No	15	57.40	1.24
D2 +vectorized	[16, 16]	4	No	15	16.63	4.29
D3 +vectorized	[16, 16]	8	No	15	9.82	7.27
D4 +conv3×3-2core	[16, 16]	8	No	20	9.18	7.77
D5 +inst-pipeline	[16, 16]	8	Yes	20	8.44	8.45
D6 +opt-tile-size	[32, 16]	8	Yes	20	5.72	13.70
D7 +opt-tile-size	[32, 32]	8	Yes	20	5.21	12.48
D8 +opt-tile-size	[32, 64]	8	Yes	20	4.99	14.30
D9 +more-cores	[32, 64]	8	Yes	40	3.16	22.58
Riallto / RAI-SW[20]	-	-	-	20	71.36	1x

The intermediate results are directly transferred between AIE cores using DMA access to the neighboring core's L1 memory. Although D1 is under-optimized, it outperforms Ryzen-AI's overlay-based approach because it caches intermediate results in the on-chip L1 cache. In contrast, the Ryzen-AI overlay utilizes more cores to accelerate a single task, writes results to slow L3 memory, and then processes the next task. In D2 and D3, vectorization is applied to OC dimension to allow SIMD processing. In D4, the second Conv3×3 task is tiled in OC dimension and mapped to 2 AIE cores to exploit tile-level parallelism. In D6-8, we adjusted the tile sizes for the image's height and width dimensions. In D9, we increased tile-level parallelism across height and width by mapping more tiles to AIE cores, achieving a 22.58× speedup over the Ryzen-AI-SW overlay.

7 Conclusion

We present ARIES, an MLIR-based compilation flow for AIE-based reconfigurable devices. ARIES provides a programming model to exploit multi-level parallelism with higher productivity and a unified IR for automated holistic optimizations. Currently, ARIES provides an open-source infrastructure for end-to-end applications on AIE-related architectures with high performance and productivity. It also provides the opportunity for the entire community to explore more advanced DSE solutions to improve the parallelism and data movement of multi-layer applications. In addition, we plan to integrate existing PL optimizations proposed by MLIR-based frameworks including ScaleHLS [38], HIDA [39], HeteroCL [40], HeteroFlow [41] and also come up with new methodologies.

ACKNOWLEDGEMENTS – This work is supported in part by Brown University New Faculty Start-up Grant, NSF awards #2019306, #2213701, #2217003, #2324864, #2328972; ACE, one of seven centers in JUMP 2.0, an SRC program sponsored by DARPA. We thank AMD for the FPGA and software donations.

References

[1] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. The design process for Google's training chips: TPUv2 and TPUv3. *IEEE Micro*, 41(2):56–63, 2021.

[2] Alejandro Rico, Satyaprakash Pareek, Javier Cabezas, David Clarke, Baris Ozgul, Francisco Barat, Yao Fu, Stephan Münz, Dylan Stuart, Patrick Schlangen, et al. AMD XDNA™ NPU in Ryzen™ AI Processors. *IEEE Micro*, 2024.

[3] Linley Gwennap. Tenstorrent Scales AI Performance. https://tenstorrent.com/vision/tenstorrent-scales-ai-performance, 2020.

[4] Eric Mahurin. Qualocmm® Hexagon™ NPU. In *2023 IEEE Hot Chips 35 Symposium (HCS)*, pages 1–19. IEEE Computer Society, 2023.

[5] AMD. Versal Adaptive SoC AIE-ML Architecture Manual (AM020), 2024.

[6] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '23, page 153–164, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Shixin Ji, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Yiyu Shi, Deming Chen, Jason Cong, and Peipei Zhou. CHARM 2.0: Composing Heterogeneous Accelerators for Deep Learning on Versal ACAP Architecture. *ACM Trans. Reconfigurable Technol. Syst.*, aug 2024. Just Accepted.

[8] Endri Taka, Aman Arora, Kai-Chiang Wu, and Diana Marculescu. MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine. In *2023 International Conference on Field Programmable Technology*, pages 96–105, 2023.

[9] Gagandeep Singh, Alireza Khodamoradi, Kristof Denolf, Jack Lo, Juan Gomez-Luna, Joseph Melber, Andra Bisca, Henk Corporaal, and Onur Mutlu. SPARTA: Spatial Acceleration for Efficient and Scalable Horizontal Diffusion Weather Stencil Computation. In *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, page 463–476, New York, NY, USA, 2023. Association for Computing Machinery.

[10] AMD. Riallto: An exploration framework for the AMD Ryzen AI NPU. https://riallto.ai/. Accessed: 2024-09-15.

[11] AMD. MLIR-AIE: An MLIR-based AI Engine toolchain. https://xilinx.github.io/mlir-aie/. Accessed: 2024-09-15.

[12] AMD. MLIR-AIR: An MLIR-based toolchain for AMD AI Engine-enabled devices. https://xilinx.github.io/mlir-air/AIRDialect.html. Accessed: 2024-09-15.

[13] Alejandro Rico, Satyaprakash Pareek, Javier Cabezas, David Clarke, Baris Ozgul, Francisco Barat, Yao Fu, Stephan Münz, Dylan Stuart, Patrick Schlangen, et al. AMD XDNA™ NPU in Ryzen™ AI Processors. *IEEE Micro*, 2024.

[14] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx Adaptive Compute Acceleration Platform: Versal™ Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 84–93, 2019.

[15] Jinming Zhuang, Zhuoping Yang, and Peipei Zhou. High Performance, Low Power Matrix Multiply Design on ACAP: from Architecture, Design Challenges and DSE Perspectives. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

[16] Zhuoping Yang, Jinming Zhuang, Jiaqi Yin, Cunxi Yu, Alex K. Jones, and Peipei 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)*, pages 1–9, 2023.

[17] Jinming Zhuang, Zhuoping Yang, Shixin Ji, Heng Huang, Alex K. Jones, Jingtong Hu, Yiyu Shi, and Peipei Zhou. SSR: Spatial Sequential Hybrid Architecture for Latency Throughput Tradeoff in Transformer Acceleration. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '24, page 55–66, New York, NY, USA, 2024. Association for Computing Machinery.

[18] Peiyan Dong, Jinming Zhuang, Zhuoping Yang, Shixin Ji, Yanyu Li, Dongkuan Xu, Heng Huang, Jingtong Hu, Alex K. Jones, Yiyu Shi, Yanzhi Wang, and Peipei Zhou. EQ-ViT: Algorithm-Hardware Co-Design for End-to-End Acceleration of Real-Time Vision Transformer Inference on Versal ACAP Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3949–3960, 2024.

[19] Chengming Zhang, Tong Geng, Anqi Guo, Jiannan Tian, Martin Herbordt, Ang Li, and Dingwen Tao. H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 200–208, 2022.

[20] AMD. AMD Ryzen™ AI Software Stack. https://www.amd.com/en/developer/resources/ryzen-ai-software.html. Accessed: 2024-09-15.

[21] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

[22] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[23] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. Allo: A Programming Model for Composable Accelerator Design. *Proc. ACM Program. Lang.*, 8(PLDI), jun 2024.

[24] Debjit Pal, Yi-Hsiang Lai, Shaojie Xiang, Niansong Zhang, Hongzheng Chen, Jeremy Casas, Pasquale Cocchini, Zhenkun Yang, Jin Yang, Louis-Noël Pouchet, et al. Accelerator design with decoupled hardware customizations: benefits and challenges. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1351–1354, 2022.

[25] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[26] AMD. AIE vector dialect. https://xilinx.github.io/mlir-aie/AIEVecDialect.html. Accessed: 2024-09-15.

[27] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 29–38, 2013.

[28] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[29] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH computer architecture news*, 44(3):367–379, 2016.

[30] AMD. AI Engine Intrinsics User Guide (UG1078), 2024.

[31] AMD. Versal Adaptive SoC AI Engine Architecture Manual (AM009), 2024.

[32] Tuo Dai, Bizhao Shi, and Guojie Luo. WideSA: A High Array Utilization Mapping Scheme for Uniform Recurrences on ACAP. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[33] AMD. Board evaluation and management Tool. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/2273738753/Versal+Evaluation+Board+-+System+Controller+-+Update+6, 2024. Accessed: 2024-09-15.

[34] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, Geoff Lowney, Adam Herr, Christopher Hughes, Timothy Mattson, and Pradeep Dubey. T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–189, 2019.

[35] Jie Wang, Licheng Guo, and Jason Cong. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 93–104, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. 1(OOPSLA), October 2017.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[38] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 741–755, 2022.

[39] Hanchen Ye, Hyegang Jun, and Deming Chen. HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 215–230, New York, NY, USA, 2024. Association for Computing Machinery.

[40] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 242–251, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. HeteroFlow: An Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '22, page 78–88, New York, NY, USA, 2022. Association for Computing Machinery.