2026 IEEE 34th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)

The Optimal, The Fast, and The Hybrid: Automatic Placement and Routing for AIE Arrays

Hang Yan,*† James Yen,*† Rongbo Zhang,*† Andrew Boutros,§ and Vaughn Betz*
*Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada
§Department of Electrical and Computer Engineering, University of Waterloo, Toronto, Canada
{hang.yan@mail, james.yen@mail, rongboz.zhang@mail, vaughn@eecg}.utoronto.ca*, andrew.boutros@uwaterloo.ca§

***Abstract*—Most of the widely deployed deep learning (DL) workloads, such as large language models and convolutional neural networks, are relatively regular compute graphs that exhibit a high degree of compute parallelism. Therefore, they are a natural fit for spatial dataflow accelerator architectures that map computations to an array of many compute cores communicating via shared memory buffers and/or some form of flexible interconnect between them, such as circuit or packet switched networks-on-chip (NoCs). AMD's adaptive intelligent engine (AIE) arrays in both the Versal FPGAs and Ryzen NPU devices are exemplars of such architectures. Despite their high peak performance, efficiently mapping workloads to these architectures to maximize compute utilization is a challenging task. The application's compute kernels are first partitioned into logical cores that are then placed at specific physical core locations. Finally, the different types of inter-core communication resources are configured to realize efficient data movement between cores. Each of these steps is a complex optimization problem that determines the ability to find a feasible mapping and directly impacts performance results. The current programming model for AIE arrays relies on manual placement or uses greedy 2D tiling algorithms. These approaches either require significant designer effort or work only for regular 2D-structured computations, but produce poor-quality or unroutable solutions for other cases. To this end, this work presents a versatile automatic placement and routing (PnR) framework for AIE arrays. We evaluate a variety of placement algorithms that guarantee optimality or trade optimality for scalability. We also formulate routing as a modified multi-commodity flow problem that is solved using mixed-integer linear programming. To demonstrate our PnR framework, we integrate it into AMD's open-source MLIR-AIE toolchain and develop an entire benchmark suite using their programming API for end-to-end performance evaluation. Across 202 synthetic and real-world benchmarks, our PnR framework finds a legal mapping for 200 out of 202 benchmarks (a 99% success rate) compared to the 62% success rate of AMD's greedy sequential placer in the MLIR-AIE toolchain. It also reduces routing resource usage by 15% compared to manual placement followed by AMD's router. On-device end-to-end runtime measurements show that our PnR produces solutions that have a 30% speedup over AMD's placer and are only 7% slower than expert manual placement.**

I. Introduction

The prevalence of deep learning (DL) workloads in a myriad of end-user applications has fundamentally shifted the requirements for modern computing hardware both in datacenters and edge consumer devices. Workloads, such as large language models (LLMs) for complex natural language processing and convolutional neural networks for vision tasks, exhibit abundant data-level and task-level parallelism, making them ideal candidates for spatial dataflow accelerators. These architectures map computations to an array of compute cores that communicate over a flexible on-chip interconnect network [1]–[3]. The AMD adaptive intelligent engine (AIE) array architecture is one of many commercial examples that follow this paradigm. It provides a tiled grid of software-programmable compute cores interconnected by high-bandwidth networks-on-chip (NoCs) as well as shared memory buffers between neighboring tiles [4]. The AMD Ryzen AI processors integrate this AIE fabric as a specialized neural processing unit (NPU) alongside the x86 CPU, integrated GPU, and shared device memory in one heterogeneous system-on-chip (SoC) [1]. In another instance of the AIE array, it coexists with a field-programmable gate array (FPGA) fabric in the AMD Versal devices [2]. These architectures rely on explicit dataflow scheduling to eliminate the overhead of complex dynamic hardware control and enhance compute efficiency by exploiting spatial pipeline parallelism. Most DL workloads can be expressed as highly regular dataflow graphs and therefore can be naturally mapped to such spatial architectures.

Although such architectures provide tremendous compute capabilities (e.g., up to 50 TOPS for the Ryzen AI NPU [5]), efficiently utilizing this *peak performance* is a challenging task that depends mainly on the characteristics of the workload and quality of its mapping to the underlying compute fabric. First, the workload dataflow graph is *partitioned* into logical compute units or kernels, which are then *placed* (i.e., assigned) to specific physical compute cores on the device grid. Finally, the communication streams between these cores are *routed* over the available interconnect resources to maximize data movement efficiency. These mapping steps are tightly coupled; suboptimal decisions in one step can significantly impact the quality of results that can be achieved by subsequent steps or even lead to an infeasible mapping. The importance of global optimization is most evident in workloads that contain feedback loops or recurrent dependencies, such as auto-regressive attention layers in LLMs. In such cases, each additional hop through the NoC on the feedback loop from the output to the input introduces additional latency. Therefore, poor quality placement that elongates feedback paths can result in underutilized cores and severely degraded throughput, even if individual compute kernels are highly optimized. Mapping workloads to these emerging spatial dataflow architectures, although fundamentally different in its details, is analogous to the packing, placement, and routing of

†Authors contributed equally to this work.

2576-2621/26/$31.00 ©2026 IEEE
DOI 10.1109/FCCM68464.2026.00015

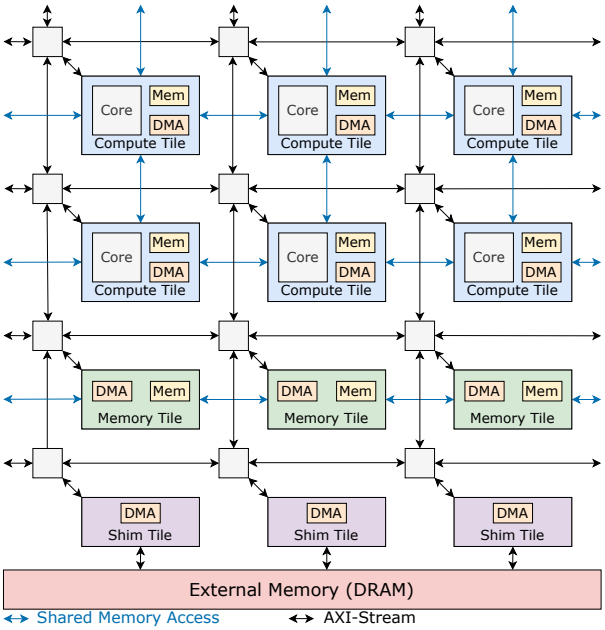

Fig. 1. AIE array architecture in the XDNA2 NPU devices.

circuit netlists on an FPGA fabric, and in this work, we develop major components of the computer-aided design (CAD) tools for solving it. In this paper, we present a versatile automatic placement and routing (PnR) framework for AIE arrays. We conduct a comprehensive study on a variety of placement algorithms that guarantee optimality as well as heuristics that trade optimality for scalability. We also formulate the mapping of inter-core communication streams to the architecture's interconnect resources as a modified multi-commodity flow problem and solve it using mixed-integer linear programming (MILP) to produce an optimal routing solution. We methodically investigate *the optimal, the fast, and the hybrid* approaches to solving this challenging CAD problem and evaluate the tradeoffs between them, conducting a direct performance comparison against AMD's open-source MLIR-AIE toolchain. Our study aims to answer the following key questions: Can optimal algorithms for mapping workloads on spatial dataflow architectures scale to the practical sizes of AIE arrays? How much quality do heuristic algorithms sacrifice to improve scalability? Although we showcase our results for AIE arrays and integrate our PnR framework into its IRON flow, our tools aim to decouple the placement and routing optimizations from specific workloads and architecture details. Our framework is capable of flexibly placing and routing a generic netlist abstraction extracted from the workload dataflow graph to a given description of the architecture's grid and routing resources. This unlocks the door for future research on the exploration of more efficient spatial dataflow architectures and CAD algorithms to target them. In summary, our key contributions include:

- Implementing a versatile open-source PnR framework[1] that can map workloads to spatial architectures with 3 types of interconnect (all found in AIE arrays): local shared memory, circuit-switched NoC, and packet-switched NoC.
- Investigating a variety of placement algorithms for spatial dataflow architectures to study the quality and scalability tradeoffs between optimal and heuristic approaches.
- Integrating our PnR framework into AMD's open-source MLIR-AIE tools to enable end-to-end performance evaluation and application deployment on Ryzen AI NPUs.
- Developing a suite of 202 synthetic and real benchmarks focusing on dataflow patterns for spatial architectures.

[1]Code can be downloaded at: https://github.com/ueqri/npu-flow.

II. Background & Related Work

A. AIE Array Architecture & Data Movement

The AMD AIE array is a two-dimensional grid of different types of tiles with a flexible interconnect fabric between them. Each *compute tile* consists of a very-long instruction word (VLIW) vector processor core, local memory, and an interconnect management unit. The VLIW core can execute 7 concurrent operations: two vector loads, one vector store, one operation on a vector processing unit that supports single-instruction multiple-data (SIMD) execution, and two operations on a scalar processing unit [6]. In the Versal adaptive SoCs, the AIE array is tightly integrated with an FPGA fabric, general-purpose Arm Contex cores, and a hardened system-level NoC that connects all SoC components and provide high-bandwidth access to external interfaces [7]. This device combines two forms of spatial computing fabric: the AIE array for high-performance arithmetic-dense workload components and the FPGA fabric for control logic, pre/post-processing, and custom compute units. The AMD XDNA NPU architecture adopts a scaled-down version of the AIE array for its consumer class CPU products [1]. In particular, the second-generation XDNA2 NPU used in this work comprises a 6×8 array. Fig. 1 illustrates the NPU AIE array, including a row of *memory tiles* that provide 512 KB of scratchpad storage ($8\times$ larger than the local memories in compute tiles) without any compute, and a row of *shim tiles* that interface with external memory.

The AIE array architecture provides multiple data movement mechanisms with unique trade-offs. The lowest-latency data movement between tiles can be achieved via **shared memory access**. This mechanism allows each tile to directly read/write data from/to the local memories of its immediate neighbors to the north, south, and east, with the west direction corresponding to its own local memory, providing the lowest-latency data access. In addition, a tile's direct memory access (DMA) engine—also referred to as a data movement accelerator in some AIE literature—can use an AXI-stream switch network for data movement to/from any non-adjacent tile. Stream-based transfers require allocating buffers in the local memories of both the sender and receiver tiles, increasing buffer usage compared to shared memory access. Using the switch network also consumes DMA channels. A memory-mapped-to-stream (MM2S) DMA channel sends data from local memory into the network, while a stream-to-memory-mapped (S2MM) channel receives data and writes it to memory. Each compute tile and shim tile has two MM2S and two S2MM channels, whereas memory tiles offer six of each. Stream switch ports can be configured in either **circuit-switched** or **packet-switched** modes. Circuit switching reserves a dedicated routing path for a single communication flow (Fig. 2b), enabling deterministic latency. Conversely, packet switching encapsulates data into packets with headers and explicit end-of-packet signaling, allowing

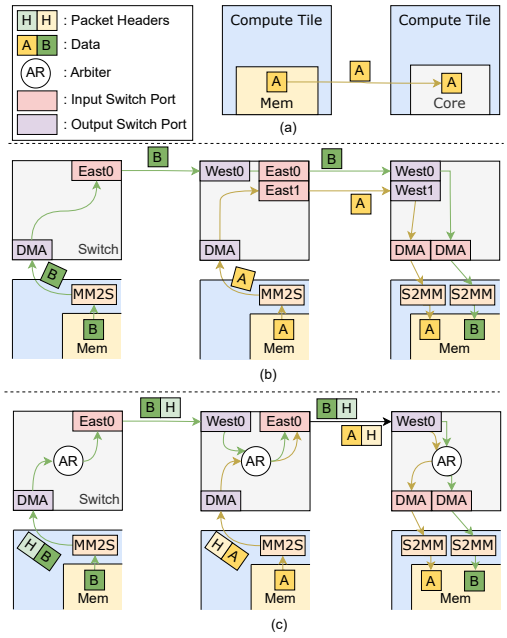

Fig. 2. Data movement types in AIE arrays: (a) shared memory access to read Input A; (b) circuit-switched and (c) packet-switched modes to transfer Output Results A and B through the NoC.

multiple flows to share routing resources (Fig. 2c). Arbiters inside each switch examine the headers and steer incoming packets to the correct output ports. While packet switching improves routing flexibility and utilization, it can introduce congestion under heavy traffic.

B. AIE Mapping Tools, Programming Models & Applications

Several prior works propose application-specific partitioning and placement strategies for AIE arrays that exploit regular structure in workloads. For example, CHARM [8] and MaxEVA [9] focus on mapping matrix multiplication and stencil-like workloads on AIE arrays. REATA [10] and CHARM 2.0 [11] offload select layers of transformer models to the AIE array. These approaches often assume that routing between placed kernels is handled automatically by downstream tools. While effective for their respective workloads, these approaches do not generalize well to other applications.

Other works present programming models that expose AIE parallelism and dataflow structure directly to the compiler. Riallto provides an end-to-end framework for machine learning with a PyTorch-like interface [12]. Ryzen-AI-SW is a software overlay that exposes higher-level runtimes, serving primarily as an execution and deployment layer [13]. Both works target only the Ryzen AI NPU and hardware details are entirely abstracted away. AMD Adaptive Data Flow (ADF) expresses an application as a static dataflow graph, but placement and routing decisions are resolved internally by the AMD Vitis toolchain and remain opaque to the user [14]. ARIES [15] offers a Python-based interface with tile-centric loop constructs optimized for GEMM-style workloads, limiting its applicability to general dataflow patterns. In contrast, IRON [16] provides lower-level control control over AIE resources and generates a multi-level-intermediate-representation-based (MLIR-based) output that exposes tile placement, buffers, and communication structure, making it well suited for this work. Its main limitations are inherited from the underlying MLIR-AIE toolchain, as described below. At the compiler level, Vitis is tightly coupled to the ADF specification and uses the proprietary `xchesscc` compiler for single-kernel compilation, and the key components of the AIE architecture and placement/routing algorithms are not publicly exposed to the user [14]. We therefore focus our comparison on MLIR-AIE. MLIR-AIE provides dialects that describe AIE architectures, memory hierarchies, and dataflow primitives. Its current placer, which we refer to as the *sequential placer* for the rest of this paper, follows a sequential greedy strategy. It assigns tiles in a column-wise order, uses shared memory access opportunistically between adjacent tiles, and routes remaining traffic using a PathFinder-style algorithm [17]. Although effective for simple designs, this placement strategy often fails to find legal mappings, as we will demonstrate later in the results section. In addition, MLIR-AIE routes both circuit-switched and packet-switched interconnects using a unified routing resource graph, without explicitly modeling or optimizing the choice between communication modes or representing shared-memory interconnects as a routing resource. It applies a single communication mode globally for stream network traffic, requiring all channels to be either circuit-switched or packet-switched, as the router cannot reason about the distinct costs and constraints of different interconnect types. In contrast, our router explicitly models multiple interconnects and their associated costs, enabling more flexible routing decisions and improved placement feasibility. MLIR-AIE uses either `xchesscc` or Peano for single-kernel compilation [18].

On the application level, although many prior works mainly focused on the use of AIE arrays for accelerating various DL inference workloads [19]–[22], other non-DL workloads that can be represented as regular dataflow graphs have also benefited from the AIE array spatial architecture and vector processing capabilities. For example, AIE arrays have been used in digital signal processing [23], radio astronomy [24], beamforming [25], computational biology [26], stencil-based weather simulation [27], [28], and option pricing [29]. In this work, we develop a suite of 202 AIE array benchmarks that have a wide variety of sizes, compute-to-communication ratios, and communication patterns complexity, and are suitable for testing our PnR framework during development and evaluating its quality of results. These are mostly synthetic benchmarks that a human designer can find an optimal solution for (to verify if the tool can also find it) and some more complex benchmarks covering linear algebra, image processing, and DL applications.

III. PnR for Spatial Dataflow Architectures

A. Framework Overview

Our work is motivated by the need for a versatile automatic PnR framework for spatial dataflow architectures in general, and more specifically for AIE arrays as an exemplar of such architectures. We develop a toolchain that (1) operates on a generic netlist abstraction extracted from the application dataflow graph, and (2) explicitly models different interconnect types in spatial dataflow architectures and incorporates the interconnect type selection tradeoffs in

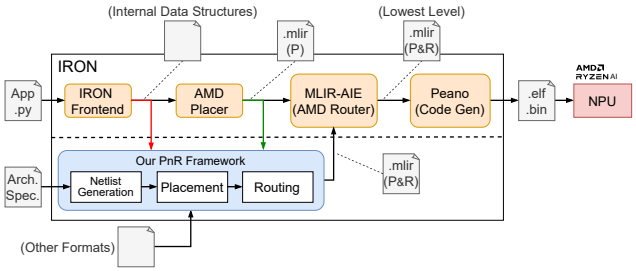

Fig. 3. Our PnR toolchain and its integration within the AMD open-source IRON implementation flow for AIE arrays in NPU devices.

the optimization process. The different components of our PnR toolchain are illustrated in Fig. 3:

1) ***Netlist Generation***: Extracts a custom netlist-like representation from the application dataflow graph. This component also takes an architecture specification input, from which it creates a device model containing both a grid of legal tile locations/types and a graph of available routing resources.
2) ***Placement***: uses MILP, simulated annealing, or a local search algorithm combined with cost estimators to optimize the mapping of *logical cores* in the application netlist to specific *physical cores*, i.e., tile locations in the grid.
3) ***Routing***: uses an MILP-based algorithm to configure the available interconnect resources for efficient data movement channels between the placed cores.

We integrate our PnR toolchain into IRON, AMD's open-source implementation flow for AIE arrays. IRON enables users to describe their application using a Python API and implements a greedy sequential placer that produces an MLIR in AIE dialect of the application including tile placement information. Then, AMD's MLIR-AIE toolchain and its internal routing engine are used to generate a lowered MLIR description including the mapping of inter-core communication streams. Finally, Peano is used for single-core code compilation followed by utilities for generating `.bin` and `.elf` files for programming the NPU device. As shown in Fig. 3, our toolchain can intercept this flow by consuming either IRON's internal data structures or the MLIR description using the AIE dialect. In the latter case, the placement solution produced by IRON's sequential placer or via manual placement is ignored by our toolchain, which produces an MLIR description including our optimized placement and routing solution. Our selected routing paths are transformed to MLIR-AIE's lower-level switchbox dialect using additional MLIR passes we implement.

B. Differences compared to FPGA Placement & Routing

In conventional FPGAs, the programmable routing provides a uniform communication fabric as blocks can be connected only through configurable switches and wire segments. Therefore, the optimization algorithms in the placement and routing engines choose only the primitive locations and the exact paths to connect between them to minimize routing cost (i.e., wirelength and congestion), without considering the impact of selecting between fundamentally different interconnect types. Even with the incorporation of hardened packet-switched NoCs in modern FPGAs [30], [31], the current CAD tools rely on the designer's choices to map latency-insensitive communication between fabric modules to the programmable routing or the NoC [32], [33].

In contrast, spatial dataflow architectures typically offer different interconnect types with different tradeoffs depending on physical locations. For example, in the case of AIE arrays, low latency communication between adjacent cores can happen through shared memory without any DMA overhead, while non-adjacent cores must utilize the AXI-stream network which consumes scarce DMA ports (only two MM2S and two S2MM channels per compute tile) and requires allocating buffers at both the sender and receiver. Using the AXI-stream network in different modes also presents different tradeoffs; circuit switching provides dedicated paths with deterministic latency but consumes exclusive resources, while packet switching enables flexible multiplexing but can introduce congestion. Thus, placement decisions determine not only how far data must travel, but also which interconnect types are available to choose from when routing each communication stream. For example, placement that separates frequently-communicating cores by even a single hop eliminates shared memory as an option, forcing the use of the less efficient streaming interconnect. Placement and routing are therefore more tightly coupled; poor placement decisions can exhaust DMA channels or create buffer conflicts that no routing algorithm can resolve.

The interconnect heterogeneity in spatial dataflow architectures creates a fundamental trade-off between *optimality* and *scalability*, which we investigate with three placement algorithms spanning this spectrum. At one extreme, an exact MILP formulation (Sec. IV) optimizes both placement and routing simultaneously, guaranteeing global optimality but at exponential cost. At the other extreme, simulated annealing (Sec. V) scales to large designs using fast proxy cost functions, but these approximations can result in sub-optimal or infeasible solutions. The third algorithm is a hybrid approach that bridges this gap (Sec. VI); it performs local heuristic search guided by an MILP oracle that exactly evaluates routing within small neighborhoods, achieving near-optimal quality with tractable runtime.

IV. The Optimal: Joint PnR MILP Formulation

The first approach we investigate in our PnR framework is an MILP-based formulation that simultaneously optimizes placement and routing decisions, guaranteeing global optimality of communication cost, congestion penalty, and buffer usage subject to all architectural constraints.

A. Routing Resource Graph Abstraction

We abstract the routing resources of spatial dataflow architectures as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where routing resource (RR) nodes $\mathcal{V}$ represent NoC switches and core interfaces, and RR edges $\mathcal{E}$ represent possible connections between them. Each physical core $p \in \mathcal{P}$ has source and sink interface nodes (p_{src}, p_{sink}). Different interconnect types are modeled as separate RR edges with distinct costs and capacities. This abstraction makes the formulation highly extensible: adding new interconnect types requires only extending the RR graph, not modifying the optimization formulation. Currently, we support three interconnect types similar to those in the AIE array architecture (see Sec. II-A): shared memory access and circuit-switched or packed-switched NoC. Each routing net

$n \in \mathcal{N}$ has a source logical core s^n and a set of target logical cores $\mathcal{T}^n$ with $|\mathcal{T}^n| \geq 1$. We also support *net chains* $\mathcal{C}$ that link consecutive data movements, enabling buffer sharing for copy, split, and merge patterns.

B. Placement Formulation

For each logical core $l \in \mathcal{L}$ and candidate physical core $p \in \mathcal{R}(l)$, where $\mathcal{R}(l)$ is the set of all physical cores that are type-compatible with logical core l, we define a decision variable $\hat{\pi}_{l,p}$, which is set if the logical core l is mapped to the physical core p (i.e., $\pi(l) = p$). This can be formulated as shown in eq. (1), and unique placement of a logical core to a single physical core is enforced using the constraint in eq. (2).

$$\hat{\pi}_{l,p} \in \{0,1\}, \quad \text{where} \quad \hat{\pi}_{l,p} = 1 \iff \pi(l) = p \tag{1}$$

$$\sum_{p \in \mathcal{R}(l)} \hat{\pi}_{l,p} = 1, \quad \forall l \in \mathcal{L} \tag{2}$$

Other constraints, e.g., to ensure physical cores are not overused, exist but are omitted for brevity.

C. Routing Formulation

We formulate the routing problem as a variant of the multi-commodity flow (MCF) problem that is used to route multiple *flow demands* between source and sink nodes in a shared network. Classical MCF assumes point-to-point (i.e., unicast) flows with fixed endpoints and permits fractional solutions [34]. The key novelty of our formulation is adapting the MC flow conservation constraint to our PnR problem by: (1) using discrete routing decision variables meaning that an edge is either fully used or not used at all, (2) enabling multicast flows that represent nets with a single source and multiple target cores, and (3) having variable flow endpoints that depend on placement decisions. For each node v in the architecture's RR graph and net n in the application netlist, our placement-dependent flow conservation constraint (i.e., the difference between the sum of incoming flows $f^n_{i,v}$ and sum of outgoing flows $f^n_{v,j}$ at an RR graph node) is defined as:

$$\sum_{(i,v)\in\mathcal{E}} f^n_{i,v} - \sum_{(v,j)\in\mathcal{E}} f^n_{v,j} = \begin{cases} -|\mathcal{T}^n| \cdot \hat{\pi}_{s^n,p} & \text{if } v = p_{src} \\ \sum_{t\in\mathcal{T}^n} \hat{\pi}_{t,p} & \text{if } v = p_{sink} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

This constraint specifies that $|\mathcal{T}^n|$ units of flow are injected at the net's source core location and one unit is absorbed at each of the net's target core locations, when the corresponding placement variables are set. This formulation naturally supports multicast nets as a flow can split at any intermediate node to reach multiple targets.

For each net n in the application netlist and edge $(i,j) \in \mathcal{E}$, we define a decision variable $x^n_{i,j}$, which is set if edge (i,j) is in the route tree of net n. The route tree structure is enforced by having at most one incoming edge of this route tree per RR node in eq. (4), and an upper bound (equal to the net fanout at this node) is set on the edge flows using eq. (5).

$$\sum_{(i,v)\in\mathcal{E}} x^n_{i,v} \leq 1, \quad \forall v \in \mathcal{V}, \forall n \in \mathcal{N} \tag{4}$$

$$f^n_{i,j} \leq |\mathcal{T}^n| \cdot x^n_{i,j}, \quad \forall (i,j) \in \mathcal{E}, \forall n \in \mathcal{N} \tag{5}$$

We also define a continuous variable $c_{i,j} \in [0, +\infty)$ that captures edge congestion (i.e., overusage) and define edge capacity constraints as shown in eq. (6).

$$c_{i,j} \geq \sum_{n\in\mathcal{N}} x^n_{i,j} - EdgeCapacity(i,j), \quad \forall (i,j) \in \mathcal{E} \tag{6}$$

D. Memory and Resource Constraints

Buffer allocation depends on the routing resources used and whether nets are chained (capable of sharing buffers and locks at an intermediate core in a multi-hop data movement). We introduce binary indicator variables that are set when specific edge types are selected (e.g., stream edges requiring endpoint buffers), then aggregate buffer demands per physical core as shown in eq. (7), where m_p denotes the total buffer consumption on physical core p.

$$m_p \leq MemoryCapacity(p), \quad \forall p \in \mathcal{P} \tag{7}$$

The total number of locks allocated must also not exceed any core's lock capacity, and is tracked with a similar set of constraints.

E. Objective Function and Scalability

In this joint formulation, we minimize the weighted sum of the routing cost, buffer usage, and penalized congestion as captured by eq. (8). The weights α, β, and γ are hyperparameters used to adjust the balance between the different terms in the cost function. $C_{i,j}$ and $P_{i,j}$ are weights that set the relative cost and overuse penalty for different RR edge types, respectively. The values of these hyperparameters and weights were empirically determined to fine-tune our PnR formulation across a subset of the benchmarks we evaluate.

$$\min \sum_{(i,j)\in\mathcal{E}} \left(\alpha \cdot C_{i,j} \cdot \sum_{n\in\mathcal{N}} x^n_{i,j} + \beta \cdot P_{i,j} \cdot c_{i,j}\right) + \gamma \sum_{p\in\mathcal{P}} m_p \tag{8}$$

The joint formulation introduces $|\mathcal{L}| \times |\mathcal{P}|$ binary placement variables coupled with $|\mathcal{N}| \times |\mathcal{E}|$ binary routing variables. The resulting MILP has an exponential worst-case complexity, making it impractical for large designs. A key observation is that the placement search space $\mathcal{P}$ dominates this complexity as device grid size increases. Therefore, to improve scalability, we seek to reduce the considered placement space. Particularly, when $\mathcal{R}(l) = \{\pi(l)\}$ contains only a single pre-assigned physical core, the formulation reduces to a pure routing problem with a given placement, creating an MILP router. In the following sections, we explore using this MILP router on a fixed placement provided by a fast heuristic placer or providing a smaller $\mathcal{R}(l)$ space via local search for the hybrid formulation.

V. The Fast: Simulated Annealing Placement

Simulated annealing (SA) [35] is a heuristic algorithm that is widely used to solve the placement problem. It starts with an initial solution and iteratively perturbs it to search for an improved solution. Every time a perturbation is performed, the cost of the new solution is evaluated to decide if the change is accepted. At the beginning of the anneal the *temperature* (T) is high resulting in a high acceptance probability and *hill climbing* exploration of inferior solutions; the temperature gradually decreases so eventually only beneficial changes are accepted. SA iteratively searches for improved solutions, making it an *anytime* algorithm that can still produce a valid solution even if interrupted before it ends. However, SA lacks strong directionality; poorly-defined cost functions or temperature schedules can cause the annealer to become trapped in local minima, leading to suboptimal solutions.

The initial temperature T_{init} controls how much exploration the annealer will do; overly high temperature wastes time exploring the solution space, while overly low temperatures

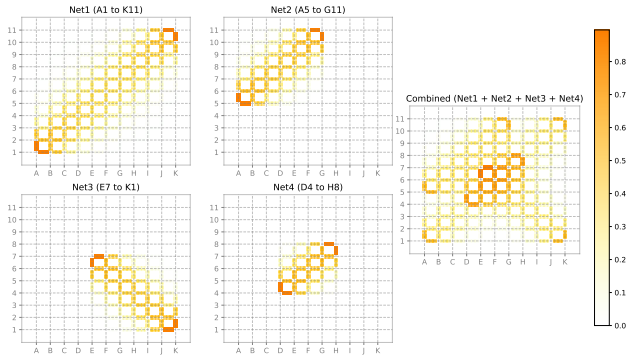

Fig. 4. Example of the channel usage probability distributions for each net and the combination of all nets.

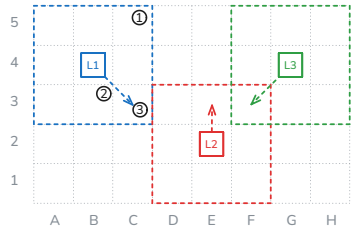

Fig. 5. Example of the LSMO algorithm. Logical cores **L1**, **L2**, and **L3** are initially placed at physical cores **B4**, **E2**, and **G4**, respectively. The dashed bounding box around each logical core denotes its neighborhood region which constrains the MILP oracle solution space. Dashed arrows indicate the next placement returned by the oracle. Nets are omitted for simplicity.

risk early convergence to a local minimum. We compute T_{nit} using the equilibrium binary search method described in [36]; at T_{init} the placement cost is expected to remain unchanged after a sequence of moves. Our temperature update schedule, adapted from [37], drops T slowly during productive phases of the annealer (approximately 50% move acceptance) and faster otherwise. The cost function C must accurately proxy routing quality while remaining computationally efficient. In this work, we evaluate three formulations: bounding-box, bounding-box with congestion cost, and MILP router cost.

1) Bounding-Box (BB) Cost: The BB cost C_{bb} consists of a routing resource utilization estimation and a legality cost.

$$C_{bb} = \sum_{n=1}^{N_{net}} q(n)\left[bb_x(n) + bb_y(n)\right] + C_{local\_overuse} \tag{9}$$

The first term in eq. (9) is inspired by VPR's wiring congestion model [37], while the second term heavily penalizes overuse of critical routing resources that can be easily determined during placement, such as DMA ports and buffers on each tile. $bb_x(n)$ and $bb_y(n)$ are the bounding-box dimensions of net n; and $q(n)$ corrects the half parameter wire-length (HPWL) underestimation for high fan-out nets.

2) BB with Congestion (BBCG) Cost: The BB cost does not capture NoC congestion, which may force long detours in the routing algorithm. We therefore augment it with a probabilistic congestion cost that estimates routing channel overuse, as in eq. (10).

$$C_{bbcg} = C_{bb} + \kappa \sum_{r}^{R_{chan}} \left[max(E(U_c(r)) - C_c(r), 0)\right] \tag{10}$$

where κ weights the congestion penalty; $E(U_c(r))$ and $U_c(r)$ are the utilization and capacity of routing channel segment r, respectively. We assume all shortest routing paths within a net BB are equally likely when computing the probability of a net using each NoC channel segment (group of parallel links at a tile); we sum over all nets to estimate the usage of each channel segment. Fig. 4 shows examples of distributions of the channel usage probability within the net BBs. By combining these distributions, we calculate the expected value of the number of nets using each channel segment.

3) MILP Router Cost: Instead of estimating NoC usage and congestion, the MILP router cost captures the routing usage and any congestion/legality issues by using the formulation defined in Sec. IV to obtain a full routing or determine no legal routing exists. This allows precise evaluation of a placement's quality but the evaluation time is extremely long when compared to the estimators above, limiting how many placement perturbations the SA algorithm can evaluate.

VI. The Hybrid: Local Search with MILP Oracle

Jointly solving placement and routing via MILP is computationally challenging for large designs due to its exponential complexity. In contrast, the heuristic SA scales effectively (with fast cost functions) though it offers no guarantees on solution quality or placement routability.

We therefore propose *LSMO*, a hybrid algorithm combining MILP's solution quality with local search's scalability, inspired by [38]. Fig. 5 illustrates the key idea: in each iteration, LSMO "asks the oracle" for the jointly optimal repositioning of all logical cores within local neighborhoods. Specifically: ① LSMO computes a neighborhood $\mathcal{R}(l, \pi)$ around the current location of each logical core. ② The joint PnR MILP from Sec. IV is invoked as an oracle, optimizing placement and routing over the restricted search space formed by the union of these neighborhoods. The oracle returns both the optimal (restricted) placement and its associated routing cost. ③ LSMO updates the current placement and records the solution if it improves upon the best found so far.

Intuitively, the MILP oracle acts as a *"gradient"* of the placement cost function, identifying the locally optimal direction of improvement. As in gradient descent, this process naturally converges to a local optimum. To escape local optima, we maintain a blacklist $\mathcal{B}$ of previously visited placements. After each iteration, the current solution is added to $\mathcal{B}$, and subsequent oracle calls are constrained to exclude blacklisted solutions. This mechanism encourages exploration beyond local minima, transforming pure descent into a form of hill-climbing.

The algorithm terminates after a fixed iteration limit or when no improvement has been observed for several consecutive iterations. LSMO can also operate in *conservative mode*, moving only one logical core per iteration for finer-grained exploration at the cost of additional oracle calls.

VII. Evaluation

This section evaluates the placement and routing algorithms introduced in Sec. IV–Sec. VI. We measure success rate, solution quality, and scalability across 202 synthetic and real world benchmarks, comparing our framework against AMD's sequential placer and AMD's router from the MLIR-AIE toolchain on the Ryzen AI NPU.

TABLE I. Benchmark Distribution across Categories

		Pipelined		Feedback Looped		Total	
		Small	Large	Small	Large		
Synthetic	Line	16	16	15	14	61	188
	Mesh	16	18	9	16	55	
	Tree	24	16	16	16	62	
Real World Application	Edge Detection	4	4	0		8	14
	GEMM	3	2			5	
	ResNet	0	1			1	
Total		63	57	40	42	202	

TABLE II. QoR of Different Placers and Routers

Placer	Router	Success Cases	Of the Common-Passing 101 Cases					
			Geomean				Arithmetic mean	
			PnR Time [s]	NPU Runtime [us]	Total Route Length	Total Buffer [KB]	# Net Shared Mem.	# Net AXI Stream
Manual	AMD	195	5	2571	19.2	1.04	6.9	8.7
	Ours	202	26	2550	19.2	1.04	6.9	8.7
AMD	AMD	125	5	3746	28.9	1.17	2.5	13
SA+MILP	Ours	160	3600	3025	23.7	1.06	6.6	12.5
SA+BB	Ours	194	52	2593	16.3	0.99	8.9	6.6
SA+BBCG	Ours	200	53	2599	16.3	0.99	8.9	6.6
LSMO	Ours	195	330	2786	17.2	1.00	8.7	6.9
MILP	Ours	164	656	2748	17.5	1.02	7.9	8.8

A. Benchmark Suite

Existing benchmarks in the AMD IRON repository [18] lack diversity in dataflow topologies. As such, we developed a benchmark suite (summarized in Tab. I) consisting of 202 workloads, including 188 synthetic cases and 14 real world applications. The synthetic benchmarks use lightweight compute kernels to isolate communication delay as the dominant bottleneck while varying netlist size and communication structure. They are designed to include similar patterns to real designs as well as stress-test patterns that challenge PnR algorithms like high-fanout nets, feedback dependency loops, and resource-constrained scenarios. They are organized into three topologies: *line*, *mesh*, and *tree*. Each topology includes both *pipeline* and *feedback-loop* variants. The pipeline variant uses uni-directional dataflow, while the feedback-loop variants introduce cycles. Benchmarks are categorized as *small* or *large* depending on whether the logical core count is below or above 50% of the AIE array size, respectively. The real world applications include edge detection, GEMM, and ResNet workloads of varying sizes.

B. Experimental Setup

All experiments were conducted on a Beelink SER9 Pro Mini PC [39] equipped with an AMD Ryzen AI 9 HX 370 processor (Strix Point NPU, 8 columns $\times$ 6 rows AIE array) and 32GB LPDDR5x-7500 RAM, running Ubuntu 25.04.

We evaluate placement algorithms using their best-found hyperparameter settings and a 3600-second timeout. For router comparisons, we provide manually placed versions of every benchmark to both AMD's router (Sec. III-A) and our MILP router (Sec. IV). Some manual designs are taken from MLIR-AIE, and others are authored following best-practice recommendations from AMD researchers with thorough inspection of quality metrics. NPU runtime is measured as wall-clock time, reporting the average of 100 iterations after 50 warm-up iterations to mitigate initial startup overhead.

C. Results

Tab. II summarizes key quality-of-result (QoR) metrics. Success rate is reported out of 202 benchmarks. Metrics for PnR (solving) time, NPU (application execution) runtime, total routing length, total buffer usage, and interconnect/communication type distributions are averaged over the 101 benchmarks that all placer-router combinations can successfully solve (i.e., common-passing cases).

Our router has a higher success rate than AMD's router. For manually placed designs, our router succeeds on all 202 benchmarks, while AMD's router fails on 7. The failures occur on designs that require a mix of circuit-switched, packet-switched, and shared-memory communication to achieve legal routing due to high fanout. AMD's router fails because it applies a single communication mode globally for AXI-stream network traffic, as discussed in Sec. II-B. Our router instead explicitly models each interconnect type as separate RR edges with distinct costs and capacities, enabling it to find legal routings that require mixed modes.

For the 195 benchmarks where both routers succeed on manual placements, routing quality metrics (route length, buffer usage, and communication mode distribution) are nearly identical. We attribute this to the abundance of routing resources on the device: stream-network switches are fully crossbar (allowing any input stream to connect to any output stream at each hop), and the horizontal and vertical directions offer six and four streams/links per routing channel, respectively. The more challenging constraint is instead the limited number of DMA ports (two MM2S and two S2MM per compute tile). These constraints strictly limit the number of traffic in the stream network, and our router's explicit modeling helps satisfy them.

Our PnR outperforms AMD's in both success rate and solution quality. Our SA with a BBCG cost function (SA+BBCG) placer achieves the highest success rate (200/202), solving 60% more benchmarks than AMD's sequential placer (125/202). Fig. 6 breaks down success rates by benchmark category. Nearly all of our placers outperform AMD's placer across most subsets, with the exception of the pure MILP placer (i.e., joint PnR approach in Sec. IV) and SA with MILP router cost (Sec. V-3), which frequently timeout on large benchmarks due to exponential runtime growth. Two design categories prove most challenging for solution legality: (1) large pipeline-communication synthetic benchmarks, where high-fanout communication and complex dataflow offset their low logical-core count, and (2) large real-world applications, where extensive multicast traffic creates challenging congestion.

On the 101 common-passing benchmarks, the SA+BBCG placer leads to the fastest NPU runtime; the average benchmark has a 30% speedup vs. its implementation with AMD's placer. We attribute this improvement primarily to feedback-loop benchmarks, whose runtime is highly sensitive to routing length—elongated feedback paths significantly increase latency. Our placer also favours usage of the low-latency shared memory between two neighboring tiles, whereas AMD's scan-line placement strategy does not fully exploit this interconnect type.

Comparison among our placement algorithms. Using manual placement as the baseline, and our MILP router to route the placement solution, SA+BBCG most closely matches manual placement's quality while achieving the

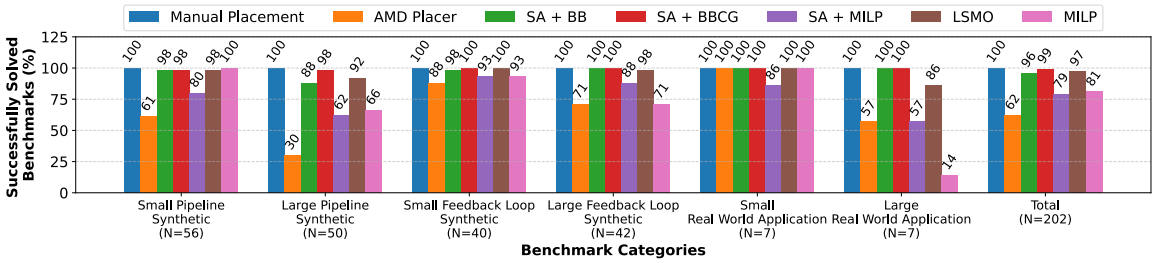

Fig. 6. Success rate of different placement algorithms across different benchmark categories.

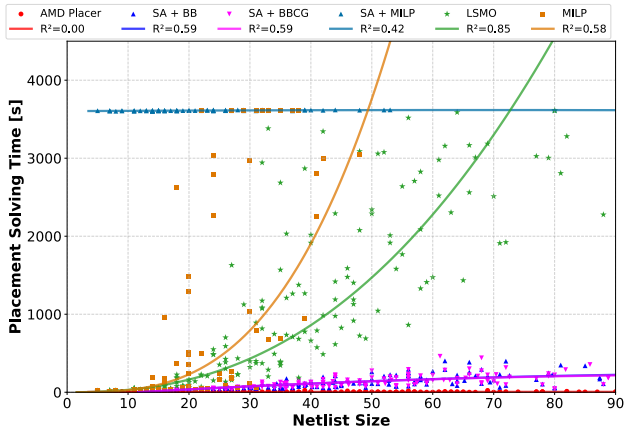

Fig. 7. Placement time vs. netlist size (number of logical cores + number of nets) of the benchmarks that complete within a 1 hour timeout.

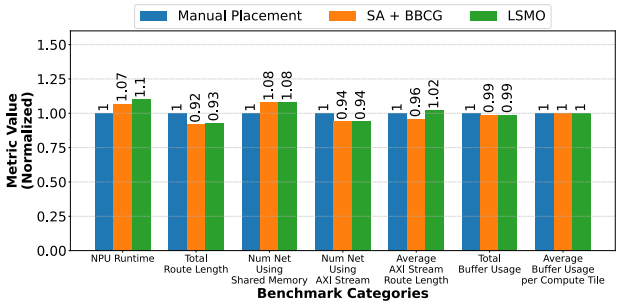

Fig. 8. QoR of the SA + BBCG and LSMO placement algorithms, normalized to manual placement (averaged over 194 common-passing cases)

highest success rate among our placers. We study scalability of the algorithms by varying netlist size. Another scalability study would be to vary the AIE array size, but we leave this for future work. As shown in Fig. 7, both the joint placement and routing MILP and SA with a cost function based on MILP routing (SA+MILP) algorithms explode in runtime as the netlist grows. For the joint place and route MILP, this stems from the large number of integer variables and constraints. For SA+MILP, the MILP router is too slow to evaluate the high volume of perturbations SA requires. The LSMO placer tames runtime by giving up global optimality and only focusing on a small region of the solution space; although the MILP router is again used as an inner loop as in SA+MILP, fewer moves are needed due to the superior moves proposed by the MILP oracle. However, LSMO still faces exponential complexity, preventing it from scaling effectively to the largest designs where SA+BB and SA+BBCG remain better options.

Fig. 8 compares the best of the algorithms with tractable runtimes, LSMO and SA+BBCG, to manual placement over the 194 benchmarks which pass in all 3 flows. Both algorithms minimize routing usage well, and use less total NoC routing than the manual placements; they are also able to use shared memory communication more often, reducing total buffer usage. SA+BBCG is able to produce better results in all categories than LSMO, and achieves close to hand-optimized manual placement NPU runtime performance (7% higher). Despite the metric dominance of SA+BBCG, LSMO provides value due to its greater architectural adaptability. While SA+BBCG relies on carefully modelled penalty functions to predict routability and legality, LSMO's generalized sub-problem approach makes it easier to port to new hardware architectures as only the router modelling needs to change.

VIII. Conclusion & Future Work

This work presents a versatile open-source PnR framework for AIE arrays that explicitly models shared memory, circuit-switched, and packet-switched interconnects—addressing the tightly coupled placement and routing problem that determines both mapping feasibility and application performance. Our MILP routing algorithm successfully routes all 202 benchmarks vs. the 195 of AMD's router. Across a range of placement algorithms, we found that an optimal MILP solution to the combined placement and routing problem does not scale. While combining SA with a constrained, local search MILP oracle (LSMO) offered an architecturally agnostic framework capable of solving the placement problem without deep hardware modelling, it is still too slow for larger problems. Ultimately, SA with detailed legality cost constraints proved superior. Our SA+BBCG approach places 200 out of 202 cases, a major improvement over the AMD placer (125/202). Furthermore, it uses 15% less routing resources compared to the manually placed solutions and the runtime of the placed and routed NPU designs is only 7% slower than manual placements.

By integrating into AMD's open-source MLIR-AIE toolchain, our tools enhance end-to-end application deployment on Ryzen AI NPUs by automating placement and routing with quality close to that of expert manual solutions. Future work includes scaling our study to larger Versal-class AIE arrays, incorporating application compute kernel partitioning for end-to-end automation, investigating machine-learning models to accelerate placement search without sacrificing quality, and generalizing the framework beyond AIE arrays to other spatial dataflow architectures.

Acknowledgement

The authors thank AMD and NSERC for funding support and Joseph Melber for helpful discussions.

REFERENCES

[1] A. Rico *et al.*, "AMD XDNA NPU in Ryzen AI Processors," *IEEE Micro*, vol. 44, no. 6, pp. 73–82, 2024.

[2] S. Ahmad *et al.*, "Xilinx First 7nm Device: Versal AI Core (VC1902)," in *IEEE Hot Chips Symposium (HCS)*, pp. 1–28, 2019.

[3] S. Lie, "Cerebras Architecture Deep Dive: First Look Inside the Hardware/Software Co-Design for Deep Learning," *IEEE Micro*, vol. 43, no. 3, pp. 18–30, 2023.

[4] AMD, *Versal Adaptive SoC AIE-ML Architecture Manual (AM020)*. AMD, 2025.

[5] B. Cohen *et al.*, "Next Generation 'Zen 5' Core," in *IEEE Hot Chips Symposium (HCS)*, pp. 1–27, 2024.

[6] AMD, *Versal Adaptive SoC AIE-ML v2 Architecture Manual (AM027)*. AMD, 2025.

[7] B. Gaide *et al.*, "Xilinx Adaptive Compute Acceleration Platform: Versal Architecture," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 84–93, 2019.

[8] J. Zhuang *et al.*, "CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, p. 153–164, 2023.

[9] E. Taka *et al.*, "MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine," in *IEEE International Conference on Field Programmable Technology (FPT)*, pp. 96–105, 2023.

[10] W. Zhang *et al.*, "REATA: An Efficient Vision Transformer Accelerator Featuring a Resource-Optimized Attention Design on Versal ACAP," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2025.

[11] J. Zhuang *et al.*, "CHARM 2.0: Composing Heterogeneous Accelerators for Deep Learning on Versal ACAP Architecture," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 17, no. 3, 2024.

[12] AMD, "Riallto: An Exploration Framework for AMD AI Engines," 2024. Accessed: 2026-01-14.

[13] AMD, "Ryzen AI Software Platform," 2024. Accessed: 2026-01-14.

[14] AMD, *Vitis Reference Guide (UG1702)*. AMD, 2025.

[15] J. Zhuang *et al.*, "ARIES: An Agile MLIR-Based Compilation Flow for Reconfigurable Devices with AI Engines," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, p. 92–102, 2025.

[16] E. Hunhoff *et al.*, "Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 85–94, 2025.

[17] Xilinx, "MLIR-AIE Documentation." https://xilinx.github.io/mlir-aie/. Accessed: 2026-01-14.

[18] Xilinx, "MLIR-AIE: An MLIR-based Toolchain for AMD AI Engine-enabled Devices." https://github.com/Xilinx/mlir-aie. Accessed: 2026-01-14.

[19] X. Jia *et al.*, "XVDPU: A High Performance CNN Accelerator on the Versal Platform Powered by the AI Engine," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[20] T. Zhang *et al.*, "A-U3D: A Unified 2D/3D CNN Accelerator on the Versal Platform for Disparity Estimation," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[21] C. Zhang *et al.*, "H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[22] P. Chen *et al.*, "Exploiting On-chip Heterogeneity of Versal Architecture for GNN Inference Acceleration," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2023.

[23] H. Yang *et al.*, "ESFA: An Efficient Scalable FFT Design Framework on Versal AI Engine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2025.

[24] V. Van Wijhe *et al.*, "Exploring the Versal AI Engines for Signal Processing in Radio Astronomy," in *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 1–10, 2024.

[25] F. Johansson *et al.*, "Beamforming of Multi-Channel Digital Radar System on System-on-Chip," 2022.

[26] G. Roks *et al.*, "Accelerated Phylogenetics on the AMD Versal Adaptive SoC," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 18, no. 3, pp. 1–26, 2025.

[27] G. Singh *et al.*, "SPARTA: Spatial Acceleration for Efficient and Scalable Horizontal Diffusion Weather Stencil Computation," in *ACM International Conference on Supercomputing (ICS)*, pp. 463–476, 2023.

[28] N. Brown, "Exploring the Versal AI Engines for Accelerating Stencil-based Atmospheric Advection Simulation," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 91–97, 2023.

[29] M. Klaisoongnoen *et al.*, "Evaluating Versal AI Engines for Option Price Discovery in Market Risk Analysis," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 176–182, 2024.

[30] I. Swarbrick *et al.*, "Network-on-Chip Programmable Platform in Versal ACAP Architecture," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 212–221, 2019.

[31] Achronix Corp., "Speedster7t Network on Chip User Guide (UG089)," 2019.

[32] S. Srinivasan *et al.*, "Placement Optimization for NoC-Enhanced FPGAs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 41–51, 2023.

[33] M. A. Elgammal *et al.*, "VTR 9: Open-Source CAD for Fabric and Beyond FPGA Architecture Exploration," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 18, no. 3, 2025.

[34] R. Ahuja *et al.*, *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993.

[35] S. Kirkpatrick *et al.*, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[36] J. Rose *et al.*, "Temperature Measurement and Equilibrium Dynamics of Simulated Annealing Placements," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 9, no. 3, pp. 253–259, 1990.

[37] V. Betz *et al.*, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 213–222, 1997.

[38] A. Cohen *et al.*, "Local Search with a SAT Oracle for Combinatorial Optimization," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 87–104, 2021.

[39] Beelink, "Beelink SER9 Pro AMD Ryzen™ AI 9 HX 370." https://www.bee-link.com/products/beelink-ser9-ai-9-hx-370. Accessed: 2026-01-14.