

From Loop Nests to Silicon: Mapping AI Workloads onto AMD NPUs with MLIR-AIR

ERWEI WANG, SAMUEL BAYLISS, ANDRA BISCA, ZACHARY BLAIR, SANGEETA CHOWDHARY, KRISTOF DENOLF, JEFF FIFIELD, BRANDON FREIBERGER, ERIKA HUNHOFF, PHIL JAMES-ROXBY, JACK LO, JOSEPH MELBER, STEPHEN NEUENDORFFER, EDDIE RICHTER, ANDRÉ RÖSTI, JAVIER SETOAIN, GAGANDEEP SINGH, ENDRI TAKA, PRANATHI VASIREDDY, ZHEWEN YU, NIANSONG ZHANG, and JINMING ZHUANG, Research and Advanced Development, AMD, USA

General-purpose compilers abstract away parallelism, locality, and synchronization, limiting their effectiveness on modern spatial architectures. As modern computing architectures increasingly rely on fine-grained control over data movement, execution order, and compute placement for performance, compiler infrastructure must provide explicit mechanisms for orchestrating compute and data to fully exploit such architectures. We introduce MLIR-AIR, a novel, open-source compiler stack built on MLIR that bridges the semantic gap between high-level workloads and fine-grained spatial architectures such as AMD’s NPUs. MLIR-AIR defines the AIR dialect, which provides structured representations for asynchronous and hierarchical operations across compute and memory resources. AIR primitives allow the compiler to orchestrate spatial scheduling, distribute computation across hardware regions, and overlap communication with computation without relying on *ad hoc* runtime coordination or manual scheduling. We demonstrate MLIR-AIR’s capabilities through two case studies: matrix multiplication and the multi-head attention block from the LLaMA 2 model. For matrix multiplication, MLIR-AIR achieves up to 78.7% compute efficiency and generates implementations with performance almost identical to state-of-the-art, hand-optimized matrix multiplication written using the lower-level, close-to-metal MLIR-AIE framework. For multi-head attention, we demonstrate that the AIR interface supports fused implementations using approximately 150 lines of code, enabling tractable expression of complex workloads with efficient mapping to spatial hardware. MLIR-AIR transforms high-level structured control flow into spatial programs that efficiently utilize the compute fabric and memory hierarchy of an NPU, leveraging asynchronous execution, tiling, and communication overlap through compiler-managed scheduling.

Additional Key Words and Phrases: Compiler, dataflow architecture, hardware acceleration, machine learning, reconfigurable technology, spatial architecture.

1 Introduction

Modern computing architectures are increasingly spatial and asynchronous. They consist of many distributed compute units, partitioned memory hierarchies, and message-passing interconnects. Achieving high performance on such architectures requires precise control over computation, data movement, and execution of tasks.

Mainstream CPUs and GPUs rely on a thread-centric parallel compute model that assumes many threads will be scheduled onto a limited set of hardware resources. Programmers describe large numbers of threads, and the system (hardware for GPUs, software for CPUs) maps them to available compute units. This model has scaled effectively for decades as semiconductor technology has

Authors’ Contact Information: Erwei Wang, erwei.wang@amd.com; Samuel Bayliss, samuel.bayliss@amd.com; Andra Bisca, andra.bisca@amd.com; Zachary Blair, zachary.blair@amd.com; Sangeeta Chowdhary, sangeeta.chowdhary@amd.com; Kristof Denolf, kristof.denolf@amd.com; Jeff Fifield, jeff.fifield@amd.com; Brandon Freiburger, brandon.freiburger@amd.com; Erika Hunhoff, erika.hunhoff@amd.com; Phil James-Roxby, phil.james-roxby@amd.com; Jack Lo, jack.lo@amd.com; Joseph Melber, joseph.melber@amd.com; Stephen Neuendorffer, stephen.neuendorffer@amd.com; Eddie Richter, eddie.richter@amd.com; André Rösti, andre.rosti@amd.com; Javier Setoain, javier.setoain@amd.com; Gagandeep Singh, gagandeep.singh@amd.com; Endri Taka, endri.taka@utexas.edu; Pranathi Vasireddy, pranathi.vasireddy@amd.com; Zhewen Yu, zhewen.yu@amd.com; Niansong Zhang, nz264@cornell.edu; Jinming Zhuang, jinming_zhuang@brown.edu, Research and Advanced Development, AMD, San Jose, USA.

delivered more cores and vector units. Hardware support for more concurrent threads improves throughput and reduces compute latency.

The effectiveness of the thread-centric model depends on two assumptions: (1) that threads operate independently, and (2) that shared resources, particularly memory bandwidth, scale with compute. When these assumptions break down, due to synchronization, data dependencies, or resource contention, execution stalls. Hardware schedulers, particularly in GPUs, respond by context-switching to another group of threads (wavefronts) to maintain forward progress. However, GPUs do not guarantee *independent* forward progress across all threads. Their schedulers may allocate compute resources to a subset of runnable wavefronts, while deferring others indefinitely. This behavior introduces nondeterminism and limits software visibility into execution dynamics, an increasingly critical shortcoming for latency-sensitive or tightly coupled workloads. Despite these limitations, the thread-centric model remains widely adopted because it simplifies software development: applications are decomposed into independent tasks that communicate through shared memory, while the hardware handles data reuse and execution interleaving. However, this abstraction comes at a significant cost. Maintaining the illusion of shared memory and uniform execution requires dense interconnects, deep cache hierarchies, and complex runtime mechanisms, all of which consume energy, silicon area, and increase design complexity.

An emerging alternative is to return control to the software. A programming model that enables explicit expression of task placement, scheduling order, and inter-task data sharing allows software to better exploit spatial and temporal locality. Rather than relying on implicit reuse via caches, such a model supports deliberate coordination between compute units that are scheduled close in space and time, reducing hardware overhead while improving predictability and efficiency.

To this end, we introduce AIR, a compiler intermediate representation (IR) that exposes spatial and temporal execution structure as explicit, programmable constructs. AIR captures high level user-described data-movement and compute scheduling intent, including concurrent execution. Implemented as a multi-level intermediate representation (MLIR) [25] dialect, AIR bridges the gap between high-level programs and spatial architectures. It supports transformations that lower structured control flow into statically scheduled spatial programs, optimized for GPUs and domain-specific neural processing units (NPUs). We demonstrate AIR’s effectiveness on two representative AI workloads: matrix multiplication and the multi-head attention (MHA) block from the LLaMA 2 model [44]. Our results demonstrate that AIR produces spatially distributed schedules that overlap communication with computation, exploit locality, and minimize runtime control overhead.

1.1 Contributions

The rapid advance of artificial intelligence (AI) models, algorithms, and accelerators has driven the adoption of diverse programming tools. Some tools focus on end-user productivity, while others are aimed at optimizing the efficient implementation of AI applications on an increasingly diverse range of specialized accelerators. MLIR is a flexible compiler abstraction designed to bridge this gap by allowing progressive lowering of designs through an extensible set of dialects [26]. Users can compose operations from a range of dialects and, in general, select transformations to achieve the goal of lowering high-level programmer intent to low-level optimized implementation¹.

AIR is an MLIR dialect that contains operations to express compute scheduling and memory allocation in spatial architectures. It operates at a level of abstraction that enables portable expression of compute kernels by avoiding explicit support for vendor-specific features. Cross-generational portability and performance scalability are supported by splitting the responsibilities for scheduling compute tasks between the compiler and runtime. This enables the compiler to define tightly

¹In some applications, MLIR is used to analyze and *raise* the abstraction of operations, rather than lower them for execution.

coupled and concurrent *herds* of execution, while giving the runtime flexibility to schedule those herds on devices whose sizes vary within and across generations of accelerator hardware.

A design expressed using the AIR dialect can use a vendor-specific lowering for implementation on an accelerator as the operations included in MLIR-AIR are intended to support common features that we observe emerging in a class of spatial hardware accelerators. Programmers or compilers can use AIR to express compute groupings and data allocations spatially, and see those decisions honored in the subsequent lowering to vendor-specific implementations.

In sum, this work makes the following key contributions:

- We present AIR, a new IR implemented as an MLIR dialect that exposes spatial and temporal structure in programs. AIR enables the compiler to coordinate computation, data movement, and synchronization — capabilities that traditional thread-centric models obscure or defer to hardware. AIR is developed as a set of spatially aware abstractions that enable lowerings from high-level programs to tiled spatial hardware. AIR models spatial partitioning with `air.herd`, point-to-point communication with `air.channel`, and explicit synchronization with `air.token`. These abstractions enable the compiler to control spatial execution, without compelling the user to drop down to lower-levels of vendor-specific abstractions.
- We build a complete end-to-end compiler flow that uses AIR to lower workloads written using high-level Python frameworks to low-level code for AMD NPUs. MLIR-AIR compiles structured loop nests into efficient, spatial programs dispatched using the NPU runtime.²
- We demonstrate MLIR-AIR’s effectiveness on two representative AI workloads. MLIR-AIR produces statically scheduled programs that exploit locality, parallelism, and pipelining on tiled hardware.

MLIR-AIR is open source and modular by design. It integrates into, and composes with other dialects with the MLIR ecosystem and provides a foundation for targeting a wide range of spatial accelerators beyond AMD’s NPU.

2 Background

This section surveys recent trends in spatial hardware that inform the architectural design of modern accelerators, which motivate key requirements on modern compilers.

2.1 Trends in Spatial Hardware

In Table 1, we describe six key trends in efficient compute hardware. Taken together, these trends define a general direction in parallel hardware design, where efficient *data movement* is the driving design philosophy. Control over *where* compute operations are dispatched and *where* data is allocated are fundamental in such systems. Emphasizing the importance of physical placement in these systems, we refer to this direction in hardware design as a movement towards *spatial hardware*. These six hardware trends collectively motivate a corresponding set of *compiler* features necessary to effectively target spatial hardware.

2.1.1 Complex System Hierarchy Design reuse is extensive in semiconductor manufacturing because of the high cost of verification. Larger chip designs are often composed of multiple chiplets, that may themselves be composed of pre-verified hardware building blocks. Non-uniform performance for similar workloads can occur if a workload is assigned resources that cross spatial or hierarchical boundaries, or if the workload uses components shared at a cluster level. Interactions between spatially arranged components can be positive (e.g., components within a cluster share a level of cache hierarchy) or negative (e.g., components arbitrate for access to a limited number of ports). In order to maximize performance and minimize negative interactions, compilers and schedulers must be aware of spatial boundaries within the chip.

²<https://github.com/Xilinx/mlir-air>

Trend	Description
Complex System Hierarchy	Design reuse introduces arbitrary boundaries in systems.
Dispatch Placement	Schedulers guaranteeing locality enable resource-sharing.
Multi-root Memory Hierarchy	Devices have independent, physically distant memory channels.
Peer Memory Movement	Efficient designs are not limited to data transfer through main memory.
Data Movement Offload	Specialized DMAs coordinate efficient data movement.
Asynchronous Execution	Distinct hardware scheduled to execute independently via dependencies.

Table 1. Six hardware trends of spatial architectures.

2.1.2 Dispatch Placement Compilers and schedulers share control of decisions over placement within a spatial architecture. As such, in order to holistically optimize placement, both compilers and runtimes must be able to control or query where a scheduler allocates compute or where a memory allocator places memory. This knowledge or control would allow a compiler optimized for spatial architectures to note the desired spatial affinity of dispatched compute elements as optional or mandatory constraints on the behavior of the runtime scheduler.

2.1.3 Multi-root Memory Hierarchy Traditional compilers treat main memory as a unifying single *root* of coherency. However, many modern devices use multiple independent memory channels to increase aggregate bandwidth. The transparent hardware-based interleaving of data across these channels offers one simple mechanism for accessing this bandwidth, but in a large device, it is likely that there is a non-uniform energy and latency cost for access to these separate memory channels. These NUMA effects have previously been observed in large multi-socket CPU systems, but compilers and runtimes now have a role to play in ensuring physical affinity between memory allocation within channels and compute scheduling even within a single package.

2.1.4 Peer Memory Movement CPUs and GPUs incorporate large amounts of on-chip SRAM memory that is used as caches and/or scratchpads. Data transfer between on-chip memories can occur implicitly in the case of coherent caches, or may be explicitly orchestrated. Effective use of on-chip memories can offer lower interconnect energy, and achieve higher realized bandwidth compared to when data is fetched multiple times from external memory.

2.1.5 Data Movement Offload GPUs and NPUs increasingly feature Direct Memory Access (DMA) engines capable of offloading complex address generation from the compute datapath to improve data movement efficiency. This enables efficient pipelined use of the interconnect fabric as well as in-line reshaping and transposition of data for efficient computation.

2.1.6 Asynchronous Execution Memory and communication operations often have considerable latency. To achieve the most efficient performance, independent actors in the system (e.g., DMAs, compute units, etc.) are kept busy, using techniques to avoid stalling during the round-trip time necessary to synchronize two concurrent components. Increasingly sophisticated hardware schedulers close to those actors interpret explicitly encoded dependencies and select the next suitable thread of work for the actor to perform.

2.2 Identified Needs in Compilers

Taken together, these trends in hardware construction motivate a desire for a software model that enables user control over scheduling and memory allocation. Specifically, we see a need for a framework that:

- (1) *Exposes hardware controls over memory allocation*, allowing users to allocate memory in different levels of the memory hierarchy and in different non-uniform memory access (NUMA) domains at each level of the memory hierarchy.
- (2) *Exposes hardware controls over compute placement*, enabling users to describe units of compute that should be scheduled concurrently, enabling tightly-coupled compute elements to optimize data-sharing and local synchronization.
- (3) *Separates data movement from computation explicitly in the IR*, enabling independent scheduling and optimization of each. This decoupling allows the compiler to overlap communication with computation, and apply architecture-specific optimizations when supported by hardware.
- (4) *Enables dependency resolution close to hardware* to minimize the time taken to observe completion of a predecessor operation. This can be achieved by expressing dependencies explicitly, and supporting lowerings that target platform-specific synchronization capabilities.

3 Related Work

The past decade has seen rapid evolution in accelerator architectures for machine learning. Many of these accelerators share the key characteristics outlined in Section 2.1: explicit spatial compute and memory hierarchies, high-throughput interconnects, and programmable DMA subsystems. Examples include Google’s TPU [23], AMD’s and Intel’s Neural Processing Units (NPUs) [21, 33], Qualcomm’s AI Engine [32], GroqChip [1], Cerebras’ Wafer Scale Engine [8], and platforms from SambaNova [31]. A defining common feature of these accelerators is the spatial allocation of compute kernels to fixed hardware regions (e.g., tiles or cores), where data is communicated via explicitly programmed on-chip data-paths, often decoupled from compute [39, 41]. While architectural designs vary, a common challenge remains: enabling compilers to map high-level programs to these platforms by spatial locality, data movement, and synchronization [37].

In response, the compiler community has developed a range of spatially aware compilation frameworks that aim to bridge the gap between abstract algorithm specification and low-level hardware control. These works largely focus on flexible frontends for compiler frameworks, compile transformations that enable efficient computation, compiler techniques for targeting a broad range of accelerators, or any combination thereof. The remainder of this section highlights notable works in each category.

Frontends for Accelerator Programming. There is a large diversity of frontends for accelerator programming frameworks. IRON provides a close-to-metal interface that allows detailed and customized performance tuning [20]. In contrast, frontends that capture intent at a higher level of abstraction are useful for flexibility, reusability, and quick adaptation to new algorithms and emerging programming models. Consequentially, MLIR-AIR and other works have focused on this higher level of abstraction. For instance, Union introduces a unified hardware-software co-design ecosystem within the MLIR infrastructure [22], which supports TensorFlow, ONNX, and COMET [28] as inputs. Similarly, SODA-OPT supports various high-level languages as inputs, including Python, C++, and Fortran [2]. XLA, while originally a standalone compiler for TensorFlow and JAX, has increasingly adopted MLIR components to enhance its modularity and extensibility [36]. Both Union and SODA-OPT use MLIR internally to increase front-end flexibility; while XLA was originally a standalone compiler, it has increasingly adopted MLIR components to enhance its modularity and extensibility. ARIES [48] provides an MLIR-based flow targeting

AMD AI Engines, with a focus on providing tile-granularity programming interface. Unlike these frameworks, MLIR-AIR defines a spatially explicit intermediate representation that directly models hardware concurrency, locality, and asynchronous execution inline of MLIR IRs, uniquely striking a balance between fine-grained compiler-managed scheduling and frontend and backend flexibility.

Polyhedral Compilation for Mapping Tasks to Resources. The extraction and spatial mapping of parallelism implicit in algorithms are central to delivering high quality of results (QoR), especially for accelerators which are often composed of many parallel compute units. The polyhedral model [6] provides a formal framework for analyzing and transforming loop nests through affine access relations and schedule functions. Early efforts in this space include Vivado High-level Synthesis, which demonstrated how affine loop transformations could be applied to high-level code to generate efficient FPGA implementations [11, 38, 40, 42]. AutoSA advanced this direction by introducing a full-stack polyhedral compilation flow targeting systolic arrays on FPGAs [45]. It applies space-time transformations and loop tiling to generate parallel accelerator kernels that maximize throughput while respecting hardware resource constraints. More recent tools extend these capabilities across broader architectural targets. Tools like Diesel [18] and PLUTO [7] utilize the polyhedral model to automatically parallelize and optimize loop nests across multiple hardware architectures, including multicore CPUs, GPUs, and FPGAs. Polygeist further enhances the applicability of polyhedral compilation by translating C to MLIR’s Affine and SCF dialects, enabling integration with modern compiler infrastructure and reuse of polyhedral analyses with MLIR-based workflows [27]. In contrast, MLIR-AIR leverages polyhedral analyses not only for loop transformations but also to guide asynchronous scheduling and data movement, integrating these capabilities within a structured, token-based IR.

Compiler Frameworks for Diverse Spatial Accelerators. Alongside the polyhedral model, many tools such as Marvel [10] and AMOS [47] offer plug-and-play mechanisms for diverse spatial accelerator architectures. By abstracting device-specific optimizations and code generation, these tools focus on compute patterns and memory hierarchies common to spatial accelerators, facilitating seamless integration across diverse hardware generations and platforms. Moreover, when targeting reconfigurable FPGA devices, frameworks like HIDA [46] and Revet [35] enable automatic generation of Register Transfer Level (RTL) code, streamlining the hardware design process without requiring extra manual effort. In contrast, MLIR-AIR emphasizes explicit modeling of spatial scheduling and asynchronous execution within the IR itself, enabling precise control over task placement without relying on external runtime coordination or fixed hardware templates.

4 MLIR-AIR: A Novel Compiler Framework for Spatial Architectures

Modern spatial accelerators require the compiler to do more than expose parallelism, they require explicit control over placement, communication, and execution order. MLIR-AIR is built to provide that control natively.

MLIR-AIR is a novel, platform-agnostic compiler framework designed to target a wide range of spatial architectures. In this work, we focus on its instantiation for AMD NPUs — a tiled architecture optimized for high-throughput and low-latency AI computations.

As shown in Figure 1, the AMD NPU architecture consists of a two-dimensional grid of compute (⊗), memory (⊞), and shim tiles (⊞). Shim tiles form the interfacing row which connects the NPU to host memory and I/O systems. These are the only tiles that can initiate a memory transaction to the SoC memory system. Memory tiles, located adjacent to the shim tiles, provide shared memory resources accessible by compute tiles throughout the array. Compute tiles comprise the majority of the array, each integrating local memory buffers with scalar and vector engines. Every tile features a dedicated DMA engine for block data transfers (represented in buffer descriptors, or BDs) over a reconfigurable streaming interconnect. This enables localized compute-memory communication, via the streaming interconnects—and peer-to-peer data movement, via either the dedicated cascade

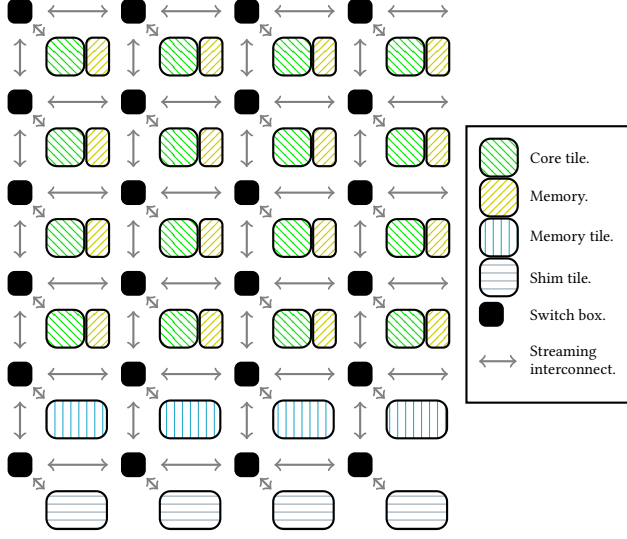


Fig. 1. AMD NPU architecture.

connections between cores or local memory shared by neighbors (②). The absence of caches, either for data or instructions, and the emphasis on computing on local tiles of data (eliminating memory access latency variation) means the architecture is characterized by extremely deterministic behaviour. Compilers designed to tile up work to fit into local memories can use the predictable behavior to construct efficient data-flow achieving high utilization.

MLIR-AIR is designed to bridge high-level algorithmic representations with the low-level spatial execution requirements of modern accelerators, such as the AMD NPU. It provides the abstractions and transformations necessary to translate structured programs into tiled, explicitly scheduled implementations. Figure 2 illustrates the MLIR-AIR compilation flow for the AMD NPU, highlighting its integration with MLIR’s ecosystem and spatial backend tools.

Algorithm-Level Programming Interface. At the compiler frontend (①), MLIR-AIR interfaces with high-level AI frameworks such as PyTorch, TensorFlow, and Triton through MLIR-compatible frontends including Torch-MLIR, TOSA, and Triton-Shared. These frameworks emit programs using structured MLIR representations that preserve loop nests, tensor operations, and affine indexing.

Frontend dialects are lowered into MLIR common components (②), including structured control flow (SCF) and linear algebra (LinAlg) dialects to provide an algorithm-friendly interface. These dialects offer C-like generic programming abstractions that preserve loop structure and tensor semantics, making AI workloads analyzable by non-domain experts. Unlike traditional low-level compilation flows that are tightly coupled to specific frontends or hardware backends, MLIR-AIR decouples emerging AI frontends from new accelerator architectures, defining a common compute model fit for the new era of spatial hardware.

Representation of Asynchronous Parallelism. At the core of MLIR-AIR is the AIR dialect (③), a set of compiler abstractions that explicitly model hardware scheduling, asynchronous execution, and interactions with the memory hierarchy. Unlike conventional IRs that assume shared memory and centralized scheduling, AIR models the constraints and opportunities of spatial systems directly in the compiler. MLIR-AIR captures fine-grained asynchronous parallelism through an asynchronous control and dataflow graph (ACDG), a directed acyclic graph that encodes MLIR operation-level dependencies sequencing computation and data movement. The ACDG is embedded directly in the MLIR-AIR IR via the production and consumption of `air.token` values, which are static

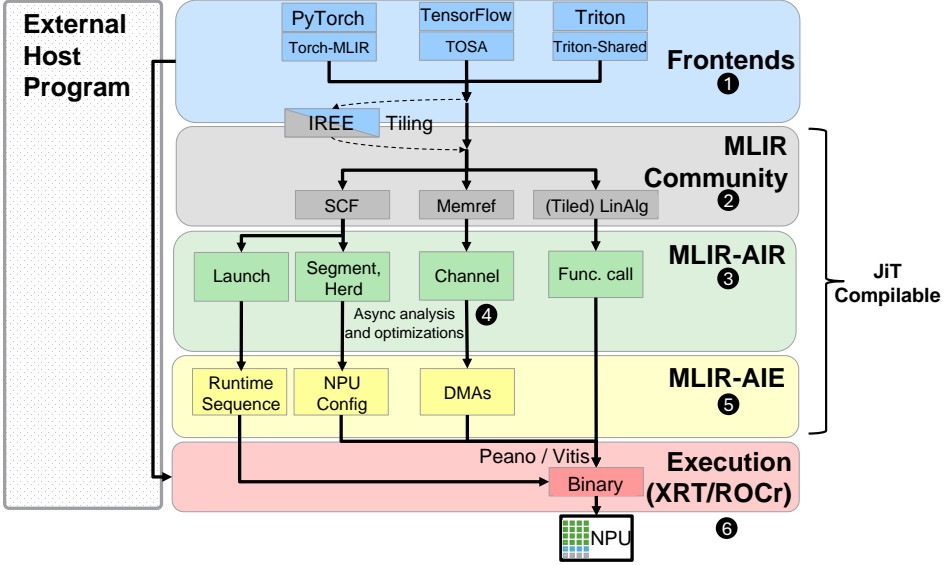


Fig. 2. MLIR-AIR stack overview.

single assignment (SSA) values encoding the read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependency types. This token-based mechanism integrates with MLIR’s native SSA dominance and verification infrastructure, enabling automatic correctness checks and transformation legality throughout the compilation process.

ACDG constructs are composable with MLIR’s structured control flow, supporting structured parallelism through tokens yielded by `scf.parallel` and AIR spatial operations (see Section 5). This allows explicit encoding of both inter- and intra-loop parallelism. Furthermore, loop-carried dependencies and pipelining opportunities are represented explicitly via tokens passed through `scf.for` iteration arguments, enabling the compiler to reason about and optimize fine-grained execution schedules, including pipeline stages and resource contention.

Decoupled Data Movement Primitives. Unlike conventional memory copy operations that couple source and destination within a single construct, MLIR-AIR introduces decoupled data movement `air.channel.put` and `air.channel.get` operations (4) to model unidirectional data transfers localized to their respective memory hierarchies. These operations are linked via a globally declared symbolic `air.channel`, which abstracts the communication path and enforces backpressure-based synchronization between asynchronous code regions. This decoupling enables fine-grained control over dataflow, allowing communication to be scheduled alongside computation when desired, or independently when beneficial for performance or modularity. The design closely aligns `air.channel` operations with their target memory scopes, allowing the compiler to reason about and optimize data movement via pattern matching of simple and localized codes.

Optimization and Performance Feedback. Beyond compilation, MLIR-AIR enhances the optimization and debugging process by providing execution traces that capture key performance metrics during hardware execution. These traces are visualized using tools like Chrome Tracing or Perfetto UI [3], allowing developers to analyze the runtime parallelism between computation and data movement. This profiling capability enables fine-grained performance evaluation, helping developers identify bottlenecks and inefficiencies in execution.

Platform-agnostic Implementation and Runtime. MLIR-AIR supports integration with multiple hardware implementation backends and runtime systems, enabling platform-agnostic compilation. The final lowered IR is consumed by platform-specific tools such as MLIR-AIE (5) for NPUs [20] or LLVM-based pipelines for CPUs and GPUs, which generate hardware-specific control and dataflow representations. These are subsequently compiled and deployed using runtime frameworks (6) such as XRT [17] or ROCr [15], ensuring compatibility with diverse hardware platforms. This modular backend integration facilitates scalable and efficient deployment while preserving the architectural flexibility of MLIR-AIR across spatially heterogeneous systems.

5 AIR Concepts

The AIR dialect provides the core primitives for expressing spatial execution semantics in MLIR-AIR. These primitives are designed to give the compiler fine-grained control over execution, concurrency, communication, and synchronization at various levels of granularity. These primitives can then be targeted by architecture-specific backends.

AIR integrates within existing compilation stacks, and transforms allow designs to be ingested from several different frontends. The AIR dialect is designed to compose with standard MLIR dialects, reusing existing dialects to describe computation and kernel. This modularity and flexibility allow developers to describe fine-grained control over execution, communication, and memory behavior while maintaining portability by decoupling these abstractions from vendor-specific hardware implementations.

We group the new operations in the AIR dialect into three categories:

- **Scheduling Constructs**, which express spatial and hierarchical parallelism across compute resources (Section 5.1).
- **Data Locality Constructs**, which represent explicit and decoupled data transfers aligned with memory hierarchy and DMA affinity (Section 5.2).
- **Synchronization Constructs**, which captures explicit operation-level and loop-carried dependencies for asynchronous execution and pipelining (Section 5.3).

Together, these abstractions allow AIR to represent concurrency control, memory placement and movement, and synchronization between many concurrent actors as explicit compiler-visible constructs.

5.1 Scheduling Constructs

AIR introduces scheduling constructs that express how computation is distributed and executed across a spatial accelerator. These constructs define task placement, launch behavior, and hardware resource partitioning, forming the foundation of spatial scheduling in AIR. The dialect includes `air.launch`, `air.segment`, and `air.herd` operations, which are hierarchically composed: an `air.launch` may contain multiple `air.segment` operations, each of which may dispatch one or more `air.herd` operations.

5.1.1 `air.launch` The `air.launch` operation defines a region of computation to be offloaded from the host processor to the accelerator. It is designed as a construct to support portability and scaling by selecting groups of operations whose dispatch may be deferred to a *runtime* scheduler. It groups together compute, communication, and synchronization operations into a single launch unit, which are scheduled at runtime.

The optional iteration space attribute attached to an `air.launch` operation describes a set of unique instances of the region body that the runtime scheduler is delegated to manage. Those unique instances must be permutable, *i.e.*, a completely parallel schedule is legal, and instances within the `air.launch` iteration space must not rely on observing the effects of any other instance.

To improve the effectiveness of compile-time optimization, we assume the compiler is free to hand off multiple different variants of the compiled `air.launch` operation to the runtime, each

enabling optimized dispatch of a parameterizable subset of the iteration space. Our lowering for AMD NPU architecture uses this freedom to offer different opportunistic granularities (e.g., one column or whole array) for the runtime to schedule work. `air.launch` also manages the lifetime of the bound resources that implement the operations hierarchically nested within its body. Once all nested operations within a launch iteration are scheduled and begin execution, the launch is able to release unused resources back to the runtime. This hierarchical management of resources ensures efficient resource utilization, especially when tasks are nested within larger compute tasks.

5.1.2 `air.segment` The body of an `air.segment` encapsulates the reservation of pool(s) of compute resources for use in scheduling the operations nested inside them. Segments can be optionally annotated with architecture-specific attributes describing the pool of resources they are reserving, when targeting backends that benefit from resource-aware scheduling. An architecture might want to define two pools of resources that have physical affinity (e.g., resources in one chiplet) so that they can ensure that the scheduler dispatches `air.herd` operations within that segment exclusively using the segment resources.

Segments have an optional grid space. This allows easy replication of resource pools. Segment instances within that space are dispatched concurrently. Other relationships between the scheduling of segments in time and space can be controlled using the synchronization constructs in Section 5.3.

5.1.3 `air.herd` The `air.herd` operation defines a group of work units that execute concurrently on a grid of physical compute units and their local memories. It contains an index space which, expressed as an affine set of worker indices, generalizes the notion of thread IDs found in traditional parallel programming models (e.g., CUDA [29] or OpenMP [9]). Each *worker* in the `air.herd` executes the same region body, but specialization is enabled by indexing: control flow and memory access patterns may diverge based on each worker’s coordinates in the `air.herd`. `air.herd` operations are scheduled atomically: they are only scheduled when resources for *all* the workers are available, and must enable independent forward progress for their individual workers. It is implied that workers are allocated as a physically local contiguous block, and lowerings may make use of the grid dimensions to lower to architecture-specific features that make use of that physical locality. The size of `air.herd` operations indicates the granularity of (concurrent) dispatch; users should be aware that certain architectures may place limits on the size of resources it can guarantee run concurrently (e.g., lowerings may fail during backend compilation if unimplementable `air.herd` operations are created).

Where multiple `air.herd` operations are included in a `air.launch`, their default behavior is to run sequentially. Programmers may use the advanced synchronization constructions in Section 5.3 to set additional constraints on `air.herd` operations to guarantee sequential or concurrent execution and consequently modify the resource requirements of the surrounding `air.launch`.

5.2 Data Locality Constructs

Spatial architectures rely heavily on local memory hierarchies and explicit DMA engines to achieve high efficiency. MLIR-AIR introduces constructs that make data locality explicit, enabling the compiler to reason about and optimize data movement across compute tiles and memory spaces.

When ingesting code from an AI framework, progressive lowerings are supported by use of MLIR MemRef types—which represent typed memory references to structured data and are assumed to reside in global memory if not explicitly annotated. Existing lowerings support explicit memory allocation and movement into at least two further levels of addressable memory hierarchy (shared cluster scratchpads and private memory local to a worker).

We support two levels of abstraction in our data movement constructs. First, to support lowering from higher-level dialects, MLIR-AIR supports an `air.memcpy` operation. Second, to provide further

control over memory movement, MLIR-AIR provides an `air.channel` operation that abstracts architecture-specific optimizations in device interconnect and specialized tensor-DMA.

5.2.1 `air.memcpy` To progressively bridge the gap between high-level memory transfer specifications and spatial hardware implementations, MLIR-AIR introduces an intermediate `air.memcpy` construct. `air.memcpy` enhances the conventional `memcpy` operation with explicit attributes for data layout and memory spaces. This allows users to indicate which levels of hierarchy they are transferring from, and to express the desire for an in-flight physical reshaping of the data, decoupling the logical layout of a `MemRef` from its physical representation in memory. This is useful because data transfer operations offer an opportunity to specialize data layout on the fly. In subsequent lowering paths, `memcpy` operations are lowered to make use of `air.channel` operations.

5.2.2 `air.channel` Many modern spatial accelerators, including AMD NPUs and NVIDIA Hopper GPUs, expose hierarchies of data movement engines and memory spaces that require explicit software modeling for efficient execution. To support this, MLIR-AIR introduces the `air.channel` abstraction, which represents stream-based data transfers between distinct memory regions through paired `put` and `get` operations:

- `put` operations transfer data from a source memory address in one level of the memory hierarchy onto a serialized stream, and
- `get` operations retrieve data from the stream into a destination memory address, representing a buffer in a particular level of the memory hierarchy.

These operations are placed in the code regions local to their respective memory allocations, enabling the compiler to express and optimize DMA-to-memory affinity. It captures both endpoints of the communication via a globally scoped symbolic `air.channel`, enabling an ordered and streamed data transfer. Subsequent compiler passes, detailed in Section 7.4, then decouple `air.memcpy` (Listing 1) into discrete `air.channel.put` and `air.channel.get` operations (Listing 2).

Notably, `air.channel` operation references can cross levels of the AIR construct hierarchy. For example, an `air.channel.put` operation that is the immediate child of an `air.launch` operation may push data into an `air.channel` whose consumers are more deeply nested `air.channel.get` operations inside `air.herd` and `air.segment` operations.

The `air.channel` abstraction integrates naturally with the `MemRef` dialect by adopting the same offset, size, and stride specifications for describing structured memory accesses. As shown in Listing 2, `air.channel` operations operate over structured `MemRef` views, allowing tensor access patterns to remain analyzable and composable with other MLIR transformations.

Listing 1. `air.memcpy` operation³.

```
1  air.memcpy (%y, %x)
```

Listing 2. Equivalent `air.channel` pair.

```
1  air.channel.put @chan1 (%x)
2  air.channel.get @chan1 (%y)
```

MLIR-AIR also supports broadcasting within `air.channel` operations, enabling a single source to supply data to multiple consumers without redundant resource usage. Broadcasting is explicitly specified through affine integer sets, which define mappings from input indices to sets of output indices and are specialized into `affine.if` conditions at lower levels. The detection and lowering of broadcast patterns is further discussed in Section 7.2.

Named bundles of `air.channel` symbols are supported to allow users to select a specific `air.channel` to `put/get` buffers (using a numerical index).

³For simplicity, we omit the offsets, sizes, and strides lists from this code snippet.

5.2.3 `air.herd` The `air.herd` operation not only defines parallel execution but also plays a crucial role in data locality. Since all workers in an `air.herd` run concurrently on allocated local resources—including compute units, local memories, and tile-local data movers—we can optimize data movement between them using methods such as double buffering to minimize memory latency (see Section 7.4.1). The lowering of `air.herd` to hardware platforms such as AMD NPUs ensures spatial contiguity of worker placement, allowing architecture-specific features to be exploited for efficient implementation of communications. For example, in its default setting, the physical lowering of `air.herd` operations to NPU AI Engine arrays guarantees that neighboring workers can write to the local memory of their neighboring cores. This allows an efficient specialized lowering of certain patterns of channel communication within the `air.herd`. By constraining data exchange to local resources, `air.herd` operations improve the dataflow efficiency within their scope.

5.2.4 `air.segment` As a resource management construct, `air.segment` also contributes to data locality by controlling memory partitioning and affinity constraints. Since `air.segment` operations dictate how resources are allocated and shared, they help ensure that accesses to the shared memories remain localized within their data producers and consumers, including the data movers and any `air.herd` operations within an `air.segment`. This reduces unnecessary data movement, keeping computation and memory access spatially co-located for better efficiency.

5.3 Synchronization Constructs

5.3.1 `air.token` The `air.token` construct provides fine-grained control over execution dependencies in asynchronous workloads, with the granularity tunable from coarse-grained synchronization over regions of code to fine-grained per-operation scheduling. When an operation is marked with the `async` keyword, it returns an `air.token` that signals its completion status; this `air.token` can be used to constrain the relative scheduling or placement of operations in time or space.

Synchronization using `air.token` is managed through two mechanisms. Firstly, explicit `air.wait_all` operations, can be inserted into synchronous control flow. This allow us to prevent further operation dispatch until tokens specified in the `air.wait_all` operation have signaled completion. Secondly, synchronization lists, that explicitly specify the relative scheduling of operations in time and space, describe a scheduling graph. Backend lowerings can use this graph to push dependency resolution to distributed dispatchers in hardware devices, allowing offload of groups of operations.

The compute model envisages three types of relationships between operations controlled by synchronization lists.

- *Dependency* lists encode directed edges between an operation and the predecessor operations in which it's inputs are defined. It requires that all inputs to a operation that are modified by a source operation in the dependency list are visible before the sink operation is scheduled. This is often implemented as a *happens-before* relationship to control the scheduling of operations in time.
- *Concurrency* lists constrain the scheduling of operations in space and time. Each `air.token` in a concurrency list defines an undirected edge between two operations that indicates they must be scheduled at the same time. This implies that each operation must use exclusive resources.
- *Affinity* lists constrain the scheduling of operations in space. These are lists of tokens that define undirected edges between operations that must execute using the same resources. In practice, this means these operations' time-slots must be disjoint, but the edge does not describe which operation must be scheduled first. The edges provide information on where spatial affinity could be exploited by the compiler or runtime.

Details on how MLIR-AIR automatically detects and lowers data dependencies are included in Section 7.3. AIR’s parallelism constructs, such as `air.launch`, signal `air.token` completion by behaving as a grouped asynchronous task: an `air.launch`’s `air.token` is released only when all operations within the `air.launch` have completed.

5.3.2 `air.channel` While primarily a data movement construct, `air.channel` also plays an essential role in synchronization across data movers operating on discrete memory spaces, where an `air.channel.get` operation is synchronized to an `air.channel.put` with the same `air.channel` by back pressure, as shown in Listing 2.

This synchronization abstraction—combined with asynchronous dependencies on `air.channel` actors to enforce synchronization local to each memory space—is both simple and effective: enforcement of dependencies between distributed data movers does not require complex control-flow dependencies across code regions.

6 AIR Dialect Constructs in Use

To concretely illustrate how AIR dialect constructs appear in practice, we present a simplified example of an element-wise vector addition program. This example bridges the conceptual descriptions of Section 5 and the compiler transformations detailed in Section 7.

The input program, shown in Appendix A, expresses a tiled vector addition in generic SCF using `scf.parallel` and `scf.for`. We use explicit `memref.copy` operations to move data between MemRef objects scheduled in a loop iteration. The program is agnostic to the target hardware and does not yet reflect spatial execution, memory locality, or asynchronous scheduling.

The corresponding AIR-transformed IR, shown in Listing 3, makes the spatial and asynchronous execution explicit. In this transformed version:

- The outer `scf.parallel` loop is replaced by an `air.launch` enclosing an `air.herd`, assigning each iteration to a spatial tile in a 2D compute grid.
- Temporary buffer allocations are restructured with explicit memory hierarchy annotations, and all data movement operations are rewritten using `air.memcpy` or decoupled `air.channel.put` and `air.channel.get`.
- Execution dependencies across asynchronous regions are made explicit with `air.token` values, enabling pipelined execution between data transfer and compute stages.

The next section describes the key compilation stages in this IR transform.

Listing 3. Element-wise vector add described in AIR dialect. The syntax has been simplified and reformatted for clarity of presentation; some MLIR dialect annotations and attributes are omitted.

```

1  air.channel @channel_0 [1, 2]
2  air.channel @channel_1 [1, 2]
3  air.channel @channel_2 [1, 2]
4  func.func @eltwise_add(%arg0: memref<65536xf32>, %arg1: memref<65536xf32>, %arg2: memref<65536xf32>) {
5      %0 = air.launch_async (%arg3, %arg4) in (1, 1) args(%arg7=%arg0, %arg8=%arg1, %arg9=%arg2) {
6          %1 = air.segment @eltwise_add_0_async args(%arg10=%arg7, %arg11=%arg8, %arg12=%arg9) {
7              %2 = air.channel.put_async @channel_0[0, 0] (%arg10[0, 0, 0] [32, 2, 512] [2048, 512, 1])
8              %3 = air.channel.put_async @channel_0[0, 1] (%arg10[0, 0, 1024] [32, 2, 512] [2048, 512, 1])
9              %4 = air.channel.put_async @channel_1[0, 0] (%arg11[0, 0, 0] [32, 2, 512] [2048, 512, 1])
10             %5 = air.channel.put_async @channel_1[0, 1] (%arg11[0, 0, 1024] [32, 2, 512] [2048, 512, 1])
11             %6 = air.channel.get_async @channel_2[0, 0] (%arg12[0, 0, 0] [32, 2, 512] [2048, 512, 1])
12             %7 = air.channel.get_async @channel_2[0, 1] (%arg12[0, 0, 1024] [32, 2, 512] [2048, 512, 1])
13             %8 = air.herd @herd_0_async tile (%arg13, %arg14) in (1, 2) {
14                 %async_token, %results = memref.alloc() async
15                 %async_token_4, %results_5 = memref.alloc() async
16                 %async_token_6, %results_7 = memref.alloc() async
17                 %async_token_8, %results_9 = memref.alloc() async
18                 %async_token_10, %results_11 = memref.alloc() async
19                 %async_token_12, %results_13 = memref.alloc() async
20                 %9 = air.wait_all_async deps=[%async_token_10, %async_token_12]
21                 %10:3 = scf.for %arg17 = 0 to 65536 step 4096 iter_args(%arg18 = %9, %arg19 = %async_token_12, %
22                     arg20 = %async_token_12) {
23                     %11 = air.channel.get_async deps=[%arg18, %arg20] @channel_0[%arg13, %arg14] (%results_9[] [])
24                     %12 = air.channel.get_async deps=[%arg18, %arg20] @channel_1[%arg13, %arg14] (%results_13[] [])
25                     %13 = air.wait_all_async deps=[%11, %12]
26                     %14 = scf.for %arg21 = 0 to 1024 step 1 iter_args(%arg22 = %13) {
27                         %async_token_20, %results_21 = memref.load_async deps=[%arg22] %results_9[%arg21]
28                         %async_token_22, %results_23 = memref.load_async deps=[%arg22] %results_13[%arg21]
29                         %22 = arith.addf %results_21, %results_23
30                         %async_token_24 = memref.store_async deps=[%arg22] %22, %results_11[%arg21]
31                         %23 = air.wait_all_async deps=[%async_token_20, %async_token_22, %async_token_24]
32                         scf.yield %23
33                     }
34                     %15 = air.channel.put_async deps=[%arg19, %arg18, %14] @channel_2[%arg13, %arg14] (%results_11
35                         [] [])
36                     %16 = air.channel.get_async deps=[%arg19] @channel_0[%arg13, %arg14] (%results[] [] [])
37                     %17 = air.channel.get_async deps=[%arg19] @channel_1[%arg13, %arg14] (%results_5[] [] [])
38                     %18 = air.wait_all_async deps=[%16, %17, %arg18]
39                     %19 = scf.for %arg21 = 0 to 1024 step 1 iter_args(%arg22 = %18) {
40                         %async_token_20, %results_21 = memref.load_async deps=[%arg22] %results[%arg21]
41                         %async_token_22, %results_23 = memref.load_async deps=[%arg22] %results_5[%arg21]
42                         %22 = arith.addf %results_21, %results_23
43                         %async_token_24 = memref.store_async deps=[%arg22] %22, %results_7[%arg21]
44                         %23 = air.wait_all_async deps=[%async_token_20, %async_token_22, %async_token_24]
45                         scf.yield %23
46                     }
47                     %20 = air.channel.put_async deps=[%19, %arg18] @channel_2[%arg13, %arg14] (%results_7[] [] [])
48                     %21 = air.wait_all_async deps=[%16, %17]
49                     scf.yield %15, %20, %21
50                 }
51             }
52         }
53     }
54     return
55 }
```

7 Compilation of AIR dialect to Spatial Hardware

This section builds upon the AIR constructs introduced in Section 5, and describes the core compiler optimizations in MLIR-AIR that transform high-level loop-based programs into efficient spatial implementations. These passes progressively transform generic operations into tiled subproblems, resolve dependencies for correct asynchronous execution, optimize data reuse and communication locality, and finally lower the program into hardware-executable IRs targeting AMD NPUs. The following subsections describe each stage in this process:

- **Tiling and Parallelism Mapping:** Maps high-level operations to distinct hardware tiles via loop tiling and parallel loop conversion (Section 7.1).

- **Broadcast Detection and Lowering:** Identifies and lowers data reuse patterns as affine-mapped broadcasts to reduce redundant transfers (Section 7.2).
- **Asynchronous Dependency Analysis:** Constructs fine-grained control and data dependencies represented using ACDG (Section 7.3).
- **Inferring Dataflow via `air.channel`:** Decouples memory transfers into local operations, exposing DMA-to-memory affinity for scheduling (Section 7.4).
- **Lowering to AMD NPU Targets:** Generates spatial hardware IR and runtime code for deployment on AMD NPUs (Section 7.5).

7.1 Tiling and Parallelism Mapping

Tiling identifies parallel subregions of computation and introduces structured iteration constructs to represent them. These parallel tiles form the basis for spatial parallelism analysis and schedule optimization. In MLIR-AIR, tiling is performed through a compiler pass pipeline that lowers implicit parallelism in high-level operations (e.g., from the `LinAlg` dialect) into explicit `scf.parallel` or `scf.for_all` loops. These are subsequently mapped to AIR spatial constructs such as `air.launch` and `air.herd`. The pipeline leverages upstream MLIR tiling utilities while also ensuring flexibility and compatibility when plugged into broader compiler ecosystems, including IREE [4] and Triton [30].

In Figure 3, we examine how tiling strategies in MLIR-AIR influence the scheduling efficiency for a tiled matrix multiplication ($A \times B = C$, where $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$, and $C \in \mathbb{R}^{M \times N}$) mapped to AMD NPUs. We consider a matrix multiplication problem tiled along the M and N dimensions and mapped to three spatial layouts: 1×4 , 2×2 , and 4×1 `air.herd` operations. These configurations are selected to cover a range of aspect ratios and communication patterns. In the 1×4 layout, matrix A is broadcast across the four column-aligned cores, while matrix B is privately transferred to each core via separate dataflows. The 4×1 layout exhibits the complementary pattern: B is broadcast across rows, but A must be duplicated. In contrast, the 2×2 layout enables two-dimensional reuse: A is broadcast column-wise, and B row-wise, minimizing redundant transfers. Data reuse patterns are automatically inferred by MLIR-AIR (detailed in Section 7.2).

Figure 3 presents the runtime traces of each strategy, with an assumption of equal tile sizes for A and B . In the 1×4 and 4×1 cases, imbalance in data streaming—due to one matrix requiring separate transfers—results in core stalls as execution waits for both inputs to arrive. By contrast, the 2×2 configuration shows reduced stalls and improved throughput owing to symmetric broadcast reuse on both A and B paths.

Performance bottlenecks in asymmetric tilings can be mitigated by rebalancing tile sizes to equalize data transfer volumes or by allocating additional DMA channels to heavier dataflows. The latter can be automated through MLIR-AIR’s dataflow-aware bufferization (see Section 7.4.3).

This case study highlights how MLIR-AIR enables fast tile-shape-aware schedule selection through explicit representation of data dependencies and broadcast opportunities, guiding design-space exploration for spatial platforms.

7.2 Broadcast Detection and Lowering

Once tiling has spatially partitioned the computational workload, the next optimization opportunity lies in minimizing off-chip data movement through on-chip reuse. To achieve this, MLIR-AIR introduces a systematic way to identify and optimize data broadcasting patterns in the tiled problem space. Compiler passes work in tandem to both detect opportunities and generate optimized AIR code that explicitly captures such patterns using affine maps.

7.2.1 Broadcast Detection The broadcast detection pass performs static analysis on the iteration domain of the program to discover replication patterns in data movements. When detected, the pass annotates the data movement operation with an affine set representing a projection in the

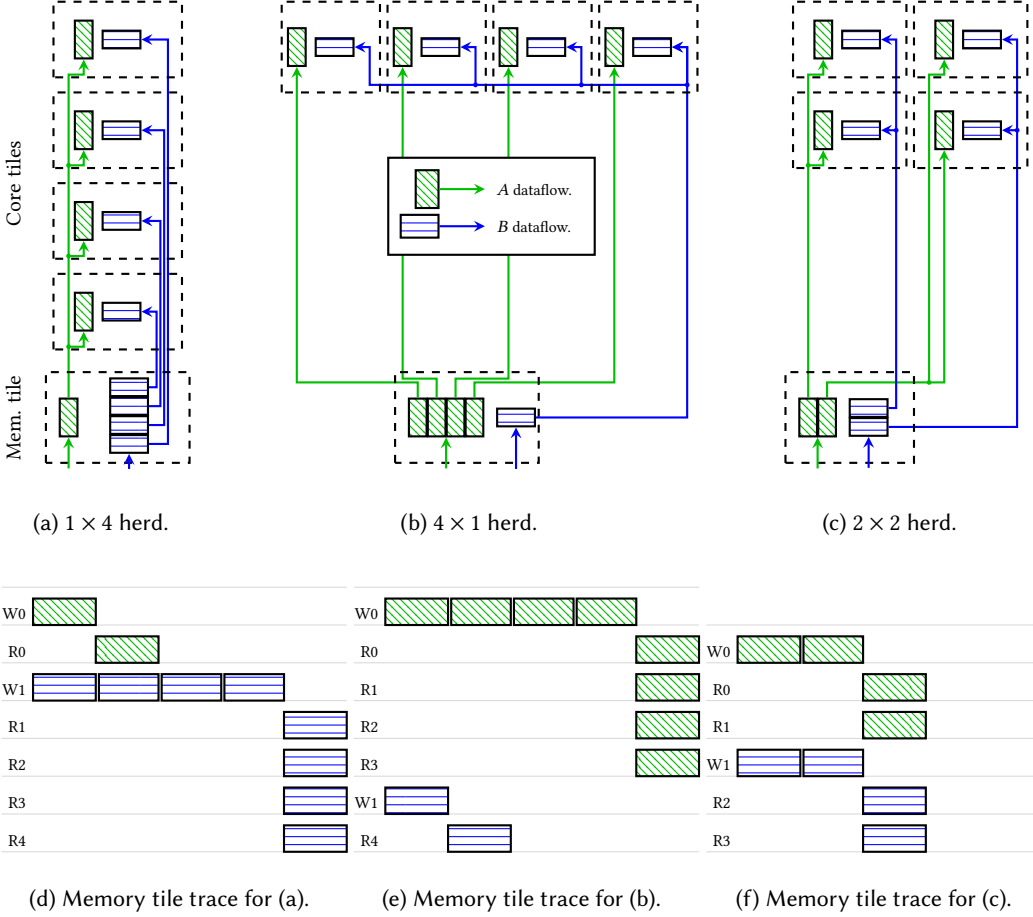


Fig. 3. Impact of tiling strategy on data movement schedule in an output-stationary matrix multiplication. See Listing 6 for its schedule described using loop nests, where the `for_all` loops `ii` and `jj` were mapped to horizontal and vertical directions in the two-dimensional `air.herd` operations, respectively.

spatial iteration domains from the sources to broadcast destinations. For example, an affine set S_0 representing a broadcast on two-dimensional spatial iterations, e.g., an `air.herd`, where an array of 4×1 `air.memcpy` sources broadcast to 4×4 destinations, has the form $\{(d_0, d_1) \in \mathbb{Z}^2 \mid \exists s_0 \in \mathbb{Z} : d_0 = s_0, 0 \leq s_0 \leq 3, 0 \leq d_1 \leq 3\}$, where the symbol s_0 and dimensions d_0 and d_1 represent the source and destination spaces, respectively.

By expressing this as an affine set in MLIR's Affine dialect, the AIR dialect retains a precise and analyzable description of the communication pattern, which remains composable with other open-source MLIR dialects thanks to the community-developed Affine dialect utilities.

7.3 Asynchronous Dependency Analysis

MLIR-AIR captures asynchronous parallelism using ACDG, represented inline of an MLIR code-base via the SSA `air.token`, which tracks the execution ordering and ensures correctness.

7.3.1 Capturing Dependencies using ACDGs In the synchronous code snippet shown in Listing 4, each data movement is implicitly blocked by the previous one, leading to a sequential schedule.

However, both DMA operations could theoretically execute simultaneously, assuming independent DMA resources. MLIR-AIR automatically analyzes memory references, identifies these implicit dependencies, and explicitly annotates synchronization tokens as shown in Listing 5.

Listing 4. Synchronous (sequential) execution.

```

1  air.memcpy (%v1, %v2)
2  air.memcpy (%v3, %v4)
3  func.call @func(%v1, %v3)

```

Listing 5. Explicit asynchronous dependencies.

```

1  %t1 = air.memcpy async (%v1, %v2)
2  %t2 = air.memcpy async (%v3, %v4)
3  %t3 = func.call async
4      deps=[%t1, %t2] @func(%v1, %v3)

```

This explicit representation clearly indicates that the compute operation must wait for *both* DMA transfers to complete before proceeding, preserving correctness. Furthermore, it also makes evident that the two DMA operations can execute in parallel, effectively leveraging multiple discrete DMA resources. In MLIR-AIR, we provide compiler passes which, driven by MLIR’s native SSA representation and dominance analysis, automatically capture the ACDG arising from the read-after-write, write-after-read and write-after-write dependencies between MLIR operations, providing robust guarantees of correctness in MLIR-AIR’s scheduling optimizations.

Loop-Carried Dependency in ACDG. In conventional compiler analysis, loop-carried dependencies are often represented using dependence polyhedra of the form $(i, j, k) \rightarrow (i', j', k')$, capturing legal source and destination iteration pairs that must respect data dependence across loops. Compilers such as PLUTO [7] and work by Baskaran *et al.* [5] typically model tiling and execution at the level of atomic tiles, where dependencies across tiles dictate scheduling order, while dependencies within a tile are assumed to be resolved independently through external memory accesses. This treatment simplifies global scheduling but leaves intra-tile parallelism and fine-grained asynchronous scheduling opportunities underexplored.

Extending beyond this model, MLIR-AIR’s ACDG captures dependencies at the operation level—both within and across loop iterations. As illustrated in Figure 4a, loop-carried dependencies are explicitly represented by passing `air.token` values (explicit synchronization handles) through the iteration arguments of `scf.for` loops, tracking per-iteration execution states. An arbitrary number of `air.token` values can be carried across iterations, each representing an independently progressing thread of execution. This enables precise modeling of parallel pipelines, race conditions, and shared resource usage, as illustrated in Figure 4b. This mechanism is particularly valuable in hardware pipelining, where producer and consumer stages can overlap in time (e.g., using ping-pong buffering; see Section 7.4.1).

Representation of Asynchronous Dependencies via `air.token` in `scf.parallel`. Similarly, MLIR-AIR supports asynchronous dependencies within structured parallel execution constructs such as `scf.parallel`. Visualized by Figure 4c, MLIR-AIR explicitly handles the synchronized initialization of parallel threads via an `air.token` passed into the initialization argument of the loop; their synchronized termination is represented explicitly via a reduction tree of `air.wait_all` barriers.

7.3.2 Reasoning using ACDGs Building on the previous section on ACDG extraction, we now describe how MLIR-AIR progressively transforms a generic loop-based program into finer-grained asynchronous schedules by analyzing and restructuring its control and data dependencies.

Figure 5 illustrates this process on an imperfect loop nest. In the original synchronous form (Figure 5a), the loop bodies imply a fully sequential dataflow in the absence of explicit parallelism annotations. Nevertheless, the underlying ACDG reveals opportunities for parallelism, as operations can be partitioned based on the memory buffers they access (annotated by colors).

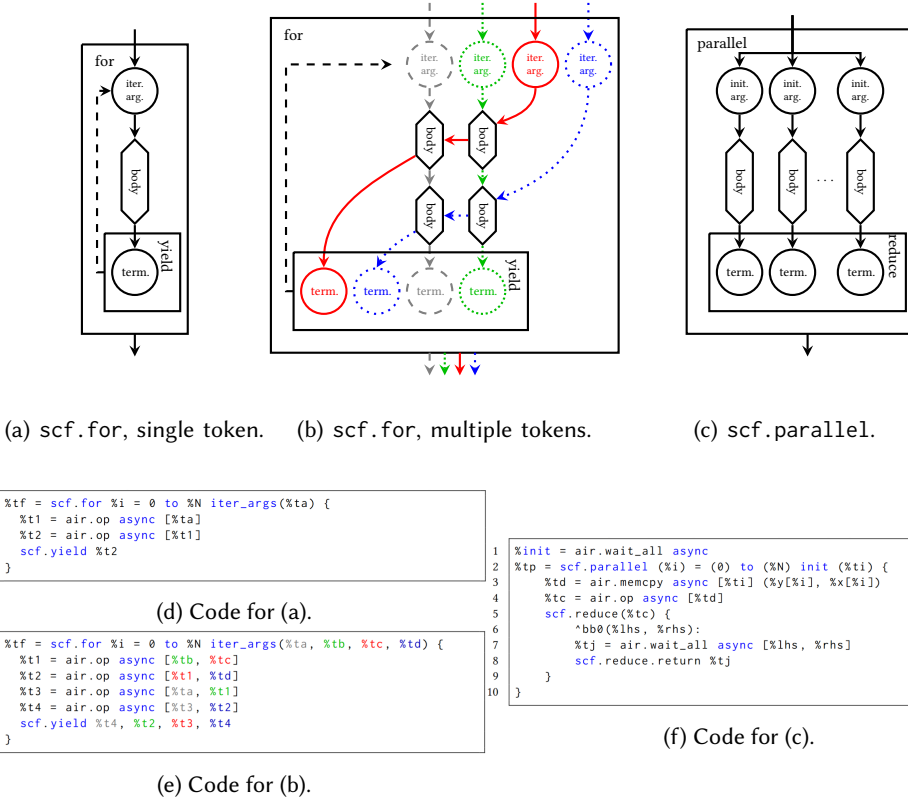


Fig. 4. Visualizations of ACDGs in loop iterations, including (a) sequentialized for loop, (b) multi-token for loop, and (c) parallel loop, and their respective MLIR-AIR specification. A circle represents an `air.token`, and a polygon represents a group of MLIR operations in the loop body. Listings (d–f) demonstrates how each ACDG is represented inline of MLIR code.

MLIR-AIR first applies asynchronous dependency analysis to construct an explicit ACDG using loop-carried `air.token` values within each loop (Figure 5b). This step exposes parallelism between sub-graphs of each loop’s body accessing distinct buffers, while preserving correctness through token synchronization.

To further expose optimization opportunities, MLIR-AIR splits the asynchronous loop nest into multiple independent nests (Figure 5c), each exclusively operating on a single memory object. This restructuring systematically uncovers and amplifies the spatial parallelism latent in generic loop-based input programs, isolating them into dataflows which facilitate compiler optimizations.

7.4 Inferring Dataflow via `air.channel`

The ACDG structure not only enables fine-grained parallelism analysis, but also serves as the foundation for identifying and scheduling data movement across disjoint memory spaces. AIR’s channel-based abstraction makes such communication patterns explicit and analyzable.

Figure 6 illustrates this transformation using ACDGs. In the pre-transformation ACDG shown in Figure 6a, a `memcpy` operation moves data from a shared buffer a into a local buffer a' , which is subsequently consumed by a compute kernel. Because `memcpy` resides within the body of the `air.herd`, the producer and consumer of a' are tightly coupled within a single hierarchical region. While this correctly expresses intra-herd dependencies, it fails to expose the fine-grained asynchronous boundary between the shared memory and local memory regions. As a result, the external

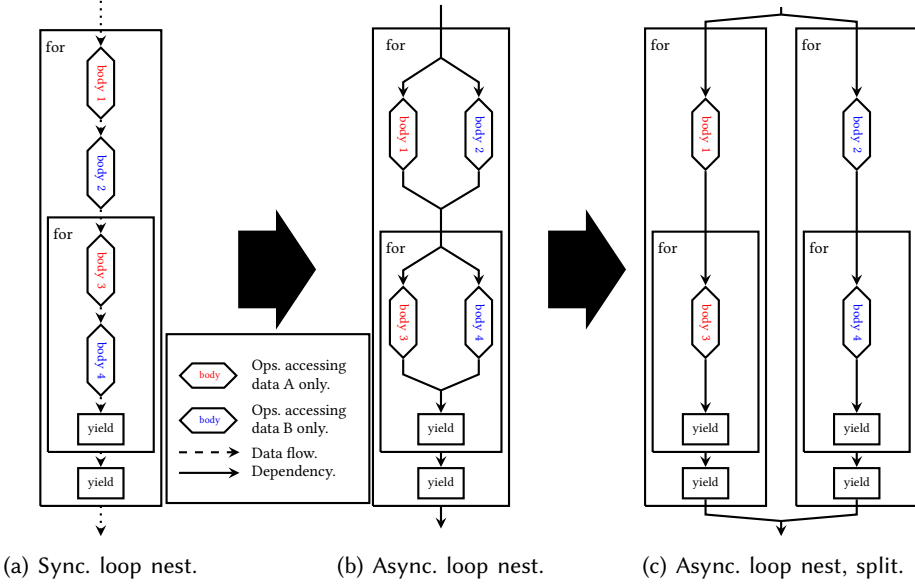


Fig. 5. Visualizations of ACDGs in loop iterations, including (a) sequentialized for loop, (b) multi-token for loop, and (c) parallel loop, and their respective MLIR-AIR specification.

thread managing *a* remains blocked until the entire `air.herd` completes—despite the fact that only the `memcpy` operation requires synchronization.

The transformed ACDG in Figure 6c resolves this limitation by replacing `memcpy` with a decoupled pair of `air.channel.put` and `air.channel.get` operations. These operations are hoisted to the respective regions associated with the source and destination memory, each integrated into its own ACDG subgraph via explicit `air.token` synchronization. To preserve the correctness of the original computation, the hoisted `put` operation must replicate the parallel semantics of the original `memcpy`; if the `memcpy` was nested within a $M \times N$ `air.herd`, then the corresponding `put` must be nested within a matching $M \times N$ `scf.parallel` loop. The dashed arrow between `air.channel.get` and `air.channel.put` represents the data stream back pressure; an overlapping schedule across two sides is enabled if stream buffering is supported in hardware. This ensures that data produced and consumed match in size across hierarchies.

This decoupling of ‘put’ from ‘get’ not only enables overlapping communication and execution but also allows the compiler to infer and instantiate multiple parallel dataflows—subject to available bandwidth and communication resources. When hardware permits, the compiler may emit parallel `air.channel` instances, increasing aggregate throughput and improving hardware utilization. In this setting, data movement is no longer serialized at the herd boundary, and bandwidth can scale with the degree of inferred parallelism.

Furthermore, the use of `air.channel` operations allows multiple data movement operations to share communication resources. This enables hardware-aware optimizations such as `air.channel` arbitration in pipelined execution (see Section 7.4.1) and `air.channel` reuse (see Section 7.4.2).

7.4.1 Capturing Hardware Pipelining with `air.channel` in ACDG Building on the fine-grained asynchronous representations introduced in Section 7.3.1 and the decoupled `air.channel` abstraction, MLIR-AIR captures hardware pipelining by leveraging the loop-carried `air.token` semantics in ACDG. By allowing multiple tokens to flow independently across iterations, MLIR-AIR models

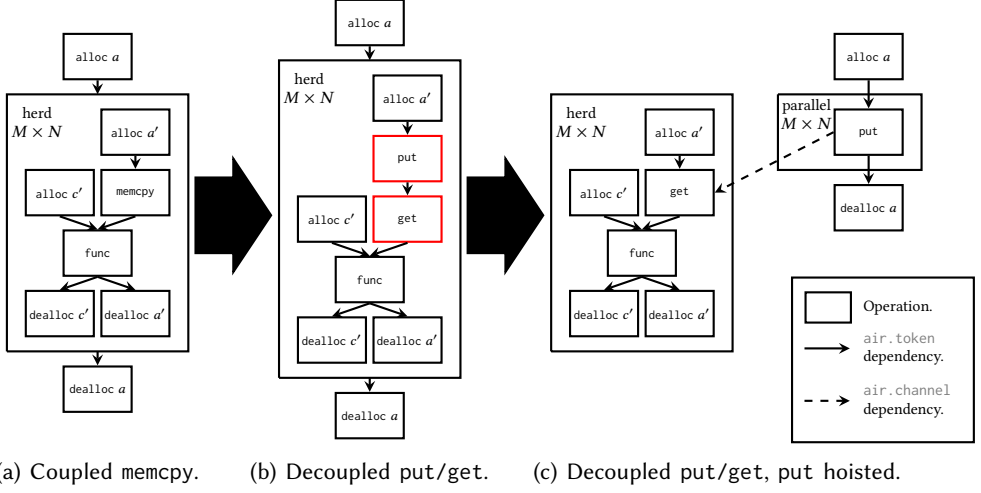


Fig. 6. Visualizations of ACDGs before and after `air.channel` decoupling.

the three key dependencies in a hardware pipeline: (i) producer-consumer data dependencies, (ii) producer-side resource contention and (iii) consumer-side resource contention, all at once.

As a motivating example, we consider two-stage pipelining, commonly referred to as ping-pong buffering. To expose pipeline stages explicitly, the loop must first be unrolled by a factor of two, corresponding to the number of stages, yielding distinct ping and pong threads for ACDG annotation. The resulting structure maps naturally to the generic ACDG with multiple loop-carried tokens shown in Figure 4b, where ping producer, ping consumer, pong producer, and pong consumer map to the four loop body subgraphs. Two of the four tokens (annotated in gray and green), represent the producer-consumer dataflow for the ping and pong stages, while the other two tokens (red and blue) capture intra-stage resource contention on the producer and consumer side, respectively. The final ACDG representing the two-stage pipeline is illustrated in Figure 7, where the flattened form highlights how each token enforces correctness across iterations.

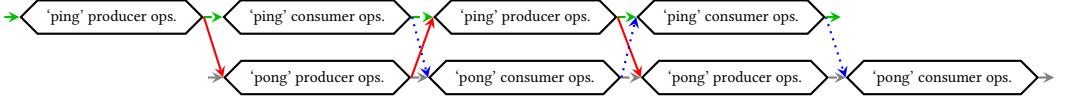


Fig. 7. Flattened ACDG showing a ping-pong buffering schedule, specialized from a generic ACDG form in Figure 4b.

To demonstrate the pipelining transformation process, we implemented a simple case study in which a data stream traverses through an AMD NPU memory tile, featuring multiple memory banks and data ports. Figure 8 shows that with ping-pong enabled, the MLIR-AIR compiler correctly identifies producer (write) and consumer (read) threads from the input loops and infers an overlapping schedule. The post-transformation runtime trace, shown in Figure 8b, confirms the expected behavior: data reads and writes execute concurrently across two buffers, validating the correctness and effectiveness of the pipelined ACDG transformation.

7.4.2 Time-multiplexed Data Movement via `air.channel` Merging The decoupled `air.channel` abstraction in MLIR-AIR enables time-multiplexed data movement by allowing multiple dataflows to reuse shared communication resources through `air.channel` merging. This is particularly valuable in scenarios where data movement hardware—such as memory ports, DMA engines, or network routing resources—is limited.

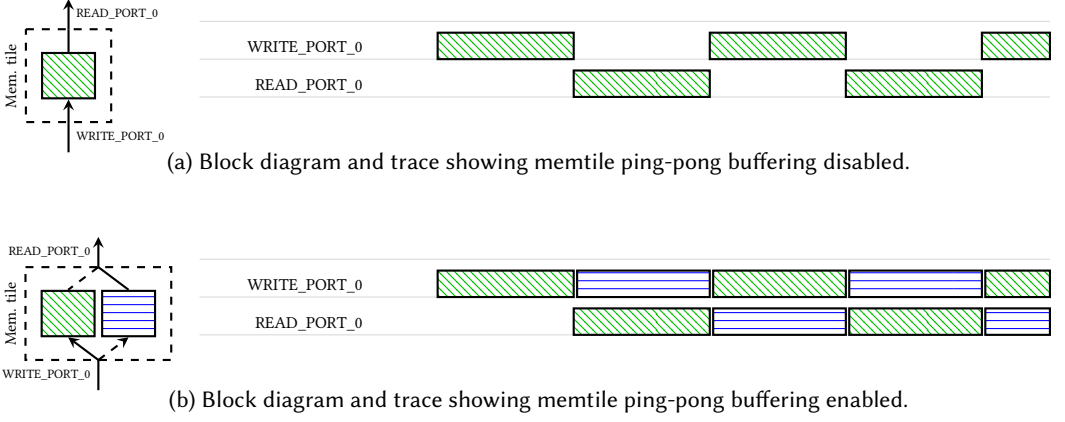


Fig. 8. A simple data streaming case study showing the effect of enabling two-stage hardware pipelining.

MLIR-AIR provides compiler passes that automatically detect opportunities for channel merging by analyzing the ACDG structure. Merging is controlled via compiler flags that specify the memory hierarchy at which merging is applied. For selected hierarchies, all merging opportunities implicit in the control flow are greedily identified and lowered.

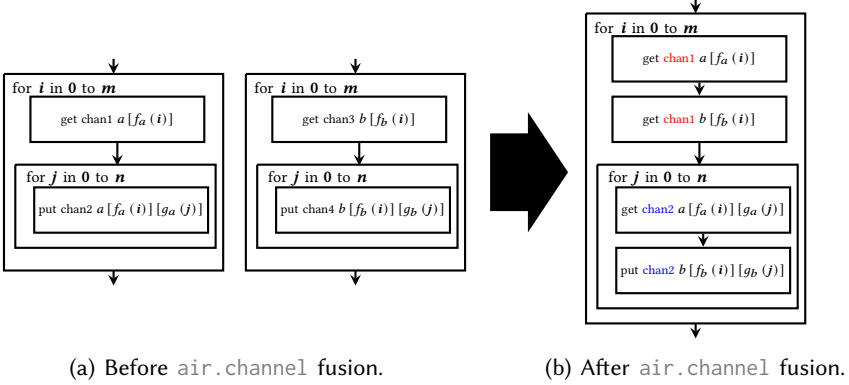


Fig. 9. Visualizations of ACDGs before and after channel merging.

Figure 9 illustrates a generic example: in Figure a, two imperfect loop nests perform channel put and get operations on separate memory objects a and b , through affine maps f_a , g_a , f_b , and g_b , respectively. Merging is permitted when the iteration domains i , j match, ensuring correctness when interleaving the data movements.

The resulting fused ACDG, shown in Figure b, sequentializes the data movements by interleaving the two loops, consolidating their use of the `air.channel` operations `@chan1` and `@chan2`.

Figure 10 further demonstrates the hardware mapping of the fused design onto an NPU memory tile, along with performance traces showing the data movement schedule. Both the original and fused designs apply pipelined execution following the scheme in Section 7.4.1. After merging, data movements are time-multiplexed, reducing contention for ports and buffers, thereby lowering resource utilization while preserving performance.

7.4.3 Parallelized Data Movement via `air.channel` Splitting While channel merging enables time-multiplexed reuse of constrained DMA resources by sequentializing data transfers, such serialization

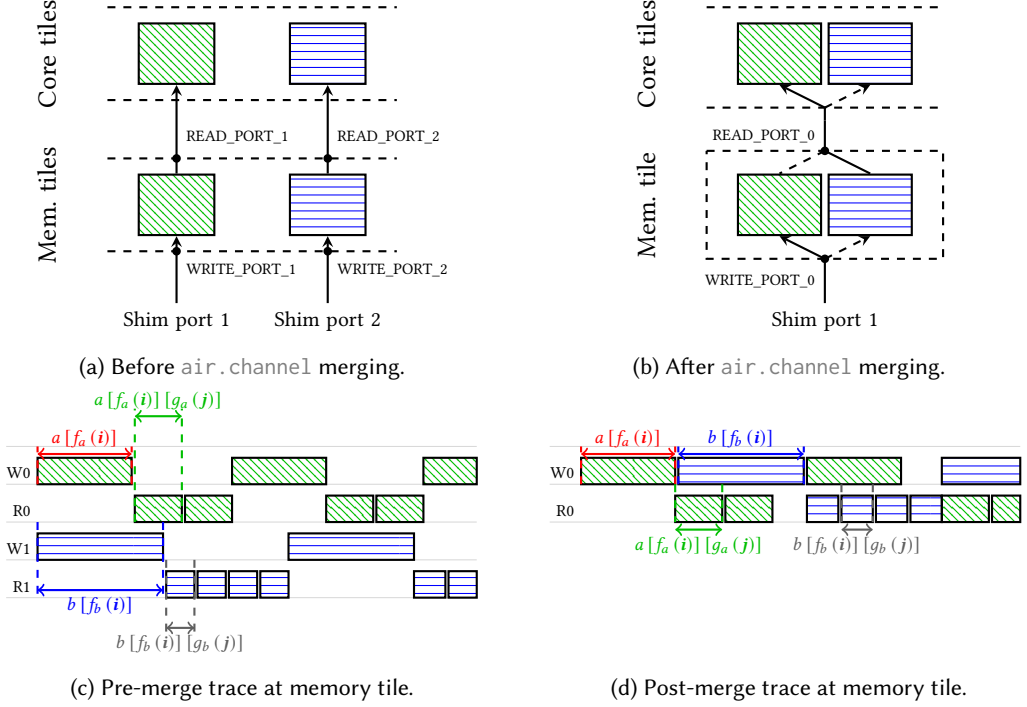


Fig. 10. Impact of `air.channel` merging on data movement parallelism and resource usage. a and b show schematic diagrams before and after `air.channel` merging. c and d show performance traces pre- and post-merging.

may limit performance when hardware availability permits greater parallelism. When DMA resources are abundant, MLIR-AIR supports an alternative strategy: exposing and exploiting data movement parallelism through MemRef splitting.

Inputs for MLIR-AIR, often from high-level IRs expressed using generic tensor abstractions, do not always consider spatial memory connectivity constraints in target architectures such as AMD NPU during bufferization, leading to degraded performance or mapping failures at implementation time. To address this, MemRef splitting performs a dataflow-aware partitioning analysis that refines buffer allocations based on the actual access patterns and hardware platform constraints.

In a common access pattern, a memory object a is read once and written to multiple outputs. Using the polyhedral representation, the read and write operations within loop nests over i and j can be represented as $a[f(i)]$ and $a[f(i)][g(j)]$, where f and g are affine maps. A concrete example of g which implies a splittable data access pattern, with $i \in \mathbb{Z}^1$, is one with dependence polyhedron $\{S0[i] \rightarrow S0[i \bmod 2]\}$, indicating two disjoint access patterns.

The affine map transformation is made possible by MLIR-AIR’s explicit representation of parallelism: by analyzing parallel `air.channel` operations and the associated asynchronous dependencies, the compiler infers the implicit parallel access patterns, transforms the affine access functions to partition independent memory accesses, and bufferizes them into smaller sub-buffers—guaranteeing parallel, conflict-free access at runtime.

In the original schedule (Figure 11a), a is naively bufferized into a single memory object, leading to sequentialized reads $a[f(i)]$ over time, regardless of available parallel memory tiles and

DMA engines. This limits parallelism across memory tiles and DMA engines, leading to reduced throughput and potential port over-utilization that can cause mapping failures.

After MemRef splitting, MLIR-AIR transforms the access maps $f \rightarrow \langle f_1, f_2 \rangle$, partitioning $a[f(i)]$ into multiple independent sub-tensors $\langle a[f_1(i)], a[f_2(i)] \rangle$ that can be allocated to separate memory tiles. This results in an optimized schedule, enabling independent and concurrent data movement across the spatial fabric.

Following this workflow, we implemented a synthetic data streaming experiment, moving data from shim ports through memory tiles to cores using a unit-stride affine access pattern. The pre-splitting hardware trace in Figure 11c shows serialization of all inbound traffic through a single tile, limiting throughput. After MemRef splitting, the post-splitting trace in Figure 11d demonstrates parallel streaming through disjoint memory tiles and shim ports, significantly improving data movement efficiency.

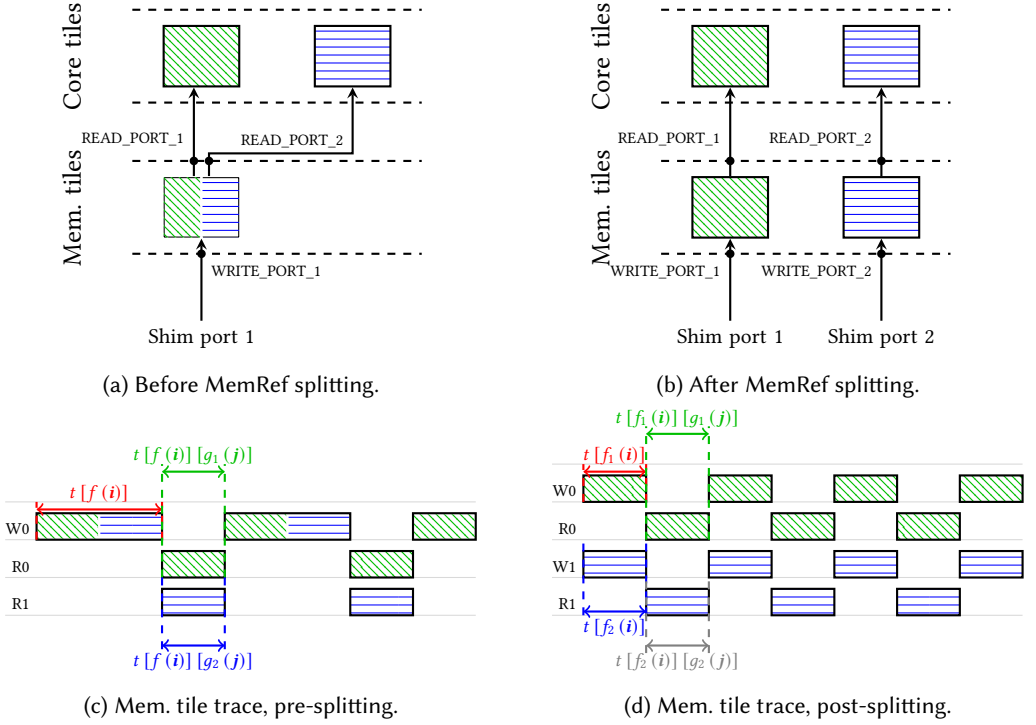


Fig. 11. Impact of MemRef splitting on data movement parallelism. a and b show schematic diagrams before and after MemRef splitting. c and d show performance traces visualizing serialization and parallelism pre- and post-splitting.

7.5 Lowering to AMD NPU Targets

With parallelism, data reuse, and communication patterns fully expressed, the AIR IR is ready to be lowered into hardware-specific representations. First, constructs in MLIR-AIR are lowered to constructs in MLIR-AIE [14]. `air.herd` operations are lowered to per-core compute kernels. `air.channel` constructs are lowered to DMA engines, BDs, and stream connections between tiles. Synchronization via `air.token` values is implemented using tile-local locks.

MLIR-AIR supports code generation targeting multiple open-source frameworks, including LLVM IR, AMD XRT, and ROCr, enabling integration into heterogeneous systems involving CPUs and GPUs. Synchronization between the hardware-specific IR and the runtime program is provided by tokens generated from the NPU hardware controller, which is lowered from `air.token` values synchronizing host operations with on-device operations.

8 Integration with AI Model Software Ecosystems

MLIR-AIR is designed to bridge the gap between high-level AI model frameworks and low-level hardware execution platforms. A key feature of MLIR-AIR is its flexible frontend integration, which allows it to ingest AI model specifications from multiple widely-used programming environments and IRs. These integrations allow developers to compile high-level AI models directly into MLIR-AIR’s asynchronous, tiled execution model, ready for targeting spatial accelerators like GPUs and AMD NPUs.

Python Integration via AIR’s Python Bindings. MLIR-AIR includes native Python bindings that expose AIR dialect operations to Python-based workflows. An example vector-add design using these bindings is shown in Appendix B. These bindings allow direct programmatic construction of AIR IR, enabling rapid prototyping and integration with AI model preprocessing, autotuning, or interactive toolchains.

PyTorch Frontend via Torch-MLIR. Through Torch-MLIR, PyTorch models are lowered into MLIR dialects which are compatible with MLIR-AIR’s tiling, scheduling, and asynchronous lowering passes. This allows MLIR-AIR to serve as a backend for PyTorch with no model rewriting, producing spatially executable kernels and runtime binaries for NPUs.

IREE Integration for Portable Deployment. MLIR-AIR interoperates with IREE by consuming its tiled intermediate MLIR representations, and producing scheduled AIR programs [4]. These can be integrated with IREE’s hardware abstraction layer (HAL), enabling deployment across heterogeneous systems where AIR-based NPUs coexist with CPU and GPU targets under a unified runtime.

Triton Frontend via Triton-Shared. AIR supports Triton through the Triton-Shared project [12], which lowers Triton IR into MLIR dialects consumable by AIR. AIR’s compilation pipeline then transforms these into hardware schedules and spatial mappings targeting AMD NPUs. This enables the reuse of GPU-oriented high-level abstractions for mapping onto NPUs. As of the date of publication, MLIR-AIR is the only compiler infrastructure that enables Triton programs to target AMD NPUs, allowing the reuse of GPU-oriented high-level abstractions for spatial architectures. The Triton-to-AIR workflow is experimental and remains under active development.

9 Design Experience and Results

We evaluate MLIR-AIR across progressively complex AI workloads to assess its abstraction efficiency, expressiveness, and performance portability. Our analysis focuses on three main dimensions: (1) programming abstraction analysis of MLIR-AIR using Halstead metrics [19], (2) performance scaling across multiple backends and hardware configurations for matrix multiplication, and (3) MLIR-AIR’s ability to express and optimize fused kernels through a case study on the LLaMA 2 MHA block. These evaluations highlight MLIR-AIR’s ability to serve as a spatial compiler abstraction that balances expressiveness and analyzability, positioning it between high-level programming models such as Triton and low-level spatial backends like MLIR-AIE.

9.1 Programming Abstraction Analysis

MLIR-AIR provides a structured, loop-based programming interface that decouples algorithm specification from hardware mapping. Developers express computation at a high level while relying on the compiler to perform hardware-aware transformations. This makes MLIR-AIR serve as an effective bridging layer between high-level programming models, such as Triton and PyTorch, and

Table 2. Difference in Halstead vocabulary, difficulty, and effort among Triton, ARIES, MLIR-AIR and MLIR-AIE, implementing the same set of common AI components to target AMD NPU (lower is better). All designs use bfloat16 data format. Green shade annotates the lowest value. Designs implemented with μ kernel called externally are annotated with ✓. While MLIR-AIE naturally shows higher complexity due to its explicit low-level programming model, MLIR-AIR bridges the gap between Triton and MLIR-AIE, offering lower effort and difficulty while maintaining spatial expressiveness.

Design	Abstraction	External μ kernel	Vocabulary		Difficulty		Effort	
			Value	×	Value	×	Value	×
matrix_scalar_add (single core)	Triton	✗	10	–	1.25	–	62.29	–
	ARIES	N/A	N/A	–	N/A	–	N/A	–
	MLIR-AIR	✗	14	1.40	1.64	1.31	112.14	1.80
	MLIR-AIE	✗	13	1.30	1.5	1.20	83.26	1.34
eltwise_binaryop	Triton	✗	12	–	1.4	–	105.40	–
	ARIES	N/A	N/A	–	N/A	–	N/A	–
	MLIR-AIR	✓	11	0.92	1.50	1.07	62.27	0.38
	MLIR-AIE	✓	23	1.92	3.579	2.56	825.67	7.83
softmax	Triton	✗	14	–	2.4	–	164.48	–
	ARIES	N/A	N/A	–	N/A	–	N/A	–
	MLIR-AIR	✓	11	0.79	1.50	0.63	62.27	0.38
	MLIR-AIE	✓	18	1.29	4.615	1.92	692.85	4.21
conv2d	Triton	✗	53	–	1.74	–	867.09	–
	ARIES	N/A	N/A	–	N/A	–	N/A	–
	MLIR-AIR	✓	11	0.21	1.50	0.86	62.27	0.072
	MLIR-AIE	✓	36	0.68	5.0	2.87	1938.72	2.24
matmul	Triton	✗	86	–	5.73	–	5410.33	–
	ARIES	✓	40	0.47	4.76	0.83	2079.31	0.38
	MLIR-AIR	✓	78	0.91	3.68	0.64	4713.03	0.87
	MLIR-AIE	✓	107	1.24	13.46	2.35	32 040.15	5.92

low-level, spatially explicit representations like MLIR-AIE, which target fine-grained hardware configurations on spatial platforms.

To evaluate the abstraction level of MLIR-AIR, we perform Halstead complexity analysis across representative AI workloads, comparing against ARIES—a compiler stack that similarly bridges high-level models to spatial hardware [48]. Halstead metrics, computed in our experiments using the open-source tool radon, quantify software complexity based on code structure that captures vocabulary, difficulty, and effort, to provide a language-agnostic measure of clarity and maintainability [24]. The Halstead metrics were evaluated across a spectrum of representative AI designs, including matrix multiplications, strided and depth-wise convolutions, nonlinear functions such as softmax and exponentiation, and trigonometric operations used in Rotary Positional Embeddings (RoPE) [43].

Table 2 reports the Halstead vocabulary, difficulty, and effort metrics across five representative workloads—each implemented using Triton [12, 30], ARIES [48], MLIR-AIR (via Python bindings), and MLIR-AIE (via IRON [20])—and highlights the key trends in abstraction efficiency and programming overhead. These examples were drawn from publicly available GitHub repositories. The workloads span a range of complexity and include both inline and externally defined μ -kernels—a templated set of compute instructions specialized to perform a task—as indicated in the ‘external

μ kernel’ column. Triton examples include μ -kernel logic inline, which inflates vocabulary and effort metrics. In contrast, MLIR-AIR and MLIR-AIE designs often invoke external kernels, resulting in more compact in-body control logic.

Despite this discrepancy in kernel inclusion, MLIR-AIR consistently maintains Halstead difficulty and effort scores within $2\times$ of Triton across the workloads. This indicates that MLIR-AIR offers a similarly accessible structured parallel programming abstraction as Triton. For smaller, single-core kernels like `matrix_scalar_add`, MLIR-AIR and MLIR-AIE show near-identical complexity. However, for complex, multi-core designs, such as in `conv2d`, and `matmul`, MLIR-AIR shows a dramatic reduction in overhead, achieving over 80% lower difficulty and effort than MLIR-AIE. This demonstrates MLIR-AIR’s strength in managing complexity as spatial parallelism increases.

ARIES presents matrix multiplication examples on their GitHub repository, which we used for comparisons in Table 2. In this example, Halstead vocabulary and effort metrics are both very low—lower than Triton—due to the reduced number of operations and operands in control logic. However, MLIR-AIR achieves a lower difficulty score, indicating AIR-based representations use simpler and more regular constructs. This result suggests that MLIR-AIR enables structured parallelism at a comparable or lower cognitive complexity than ARIES, while supporting a broader class of workloads.

These results demonstrate that MLIR-AIR effectively bridges the programming gap between the high-level Triton-style control flow and the low-level, highly explicit MLIR-AIE representation. By combining structured, tile-aware abstractions with token-based asynchronous scheduling, MLIR-AIR enables efficient spatial hardware mapping while significantly reducing the complexity developers must manage in their source code.

9.2 Performance Scaling: Mapping Matrix Multiplication to Spatial Hardware

To evaluate the ability of MLIR-AIR to generate efficient spatial compute kernels from generic loop-based programs, we examine its performance on matrix multiplication. Our experiments span a range of problem sizes (256-4096 per iteration dimension) and data formats, measured on an laptop platform featuring the AMD Ryzen AI 7840 NPU [13, 16]. We executed all MLIR-AIR and MLIR-AIE programs using the AMD XRT runtime [17], which manages binary loading, data movement, and kernel dispatch. We evaluate our MLIR-AIR-generated MLIR-AIE dialect code against MLIR-AIE’s published hand-optimized matrix multiplication implementation, which has been adopted by many recent research works as the state-of-the-art baseline for spatial execution on AMD NPUs [20, 34, 48]. The goal of this evaluation is to demonstrate that MLIR-AIR, starting from a naively specified nested loop for matrix multiplication, can produce performant implementations through a sequence of compiler transformations.

Listing 6 shows pseudocode for tiled matrix multiplication written in a generic loop-nest style, using `for` loops for sequential execution and `for_all` for spatial parallelism. Such generic representations are beneficial for portability by using an algorithm specification decoupled from hardware mapping, where AI frameworks can target MLIR-AIR without needing to provide device-specific code, demonstrating ease of frontend integration. MLIR-AIR compiles this form via a series of compilation passes presented in Section 7, which optimizes the bufferization, data movement scheduling and concurrency modeling with platform awareness.

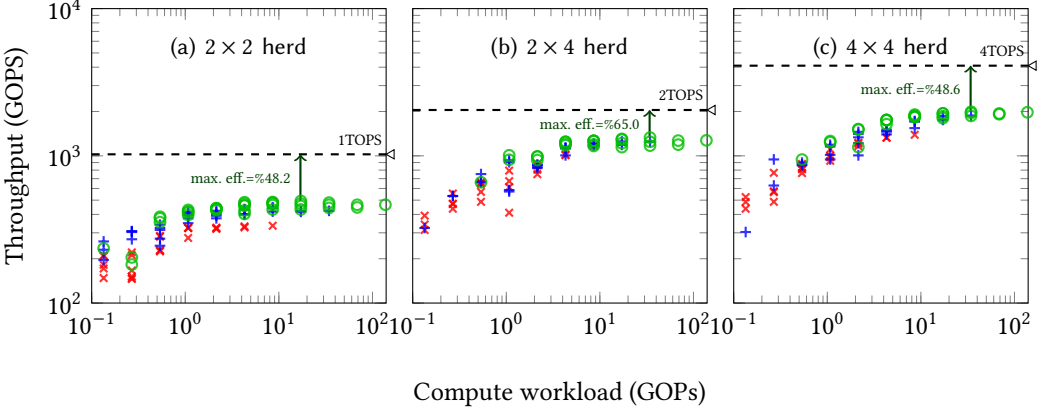


Fig. 12. Throughput versus compute workload for bfloat16 matrix multiplications, with shapes up to $M = N = K = 4k$, for AIE tile herd sized (c) 2×2 , (b) 2×4 and (a) 4×4 , respectively. Compute workload increases along the x-axis. Each point represents the maximum throughput achieved across 20 random tests, to filter out any random system and DDR access latency injected at runtime. Each color/shape reflects a distinct K , with (x), (+) and (o) annotating tests using $K = 256$, 1024 and 4096, respectively. Dotted line marks the theoretical peak compute throughput⁴ achievable for each herd of AIE cores at bfloat16 precision, and the maximum compute efficiency achieved against it is annotated with an arrow.

Listing 6. Pseudocode for a tiled output-stationary matrix multiplication that drives MLIR-AIR

```

for_all (i_outer = 0; i_outer < M; i_outer+=t_i) {
  for_all (j_outer = 0; j_outer < N; j_outer+=t_j) {
    for (k_outer = 0; k_outer < K; k_outer+=t_k) {
      for_all (ii = 0; ii < t_i; ii++) {
        for_all (jj = 0; jj < t_j; jj++) {
          for (kk = 0; kk < t_k; kk++) {
            C[ii][jj] += matmul(A[ii][kk], B[kk][jj]);
          }}}
        }
      }
    }
  }
}

```

The loop nest structure shown above implements an output-stationary schedule: each compute tile accumulates a portion of the output matrix locally across multiple input tile iterations (k loop). This is a naive but widely applicable strategy, offering high reuse of output accumulators, low communication cost for partial sums, and simple mapping to spatial arrays. However, it is only one of many possible schedules supported by MLIR-AIR, as MLIR-AIR performs schedule optimizations on generic control-flow constructs, allowing for adaptability towards different compute problems and platform constraints.

Figure 12 shows the performance of MLIR-AIR-compiled matrix multiplication kernels generated from a generic loop nest of the form shown in Listing 6, plotted as throughput (GOP/s) versus compute workload (GOPS). Three spatial hardware configurations were evaluated: 4×4 tile array (4 TOP/s peak) 2×4 tile array (2 TOP/s peak), and 2×2 tile array⁵ (1 TOP/s peak).

In this plot, the tiling sizes were fixed to $M = N = K = 64$, using the bfloat16 (bf16) data format for both input and output buffers; this tile size is chosen to fit entirely within the local memory of a single NPU tile sized 64KB. Note that this tile size was chosen heuristically and not fine-tuned; higher performance may be possible by adjusting tiling factors based on memory hierarchy and DMA burst sizes.

⁴Theoretical peak compute throughput is calculated as the maximum compute speed achievable by the specified compute units, with the assumption of infinite data movement bandwidth and zero control overhead.

⁵Herd is reshaped to occupy a single column of four AIE tiles.

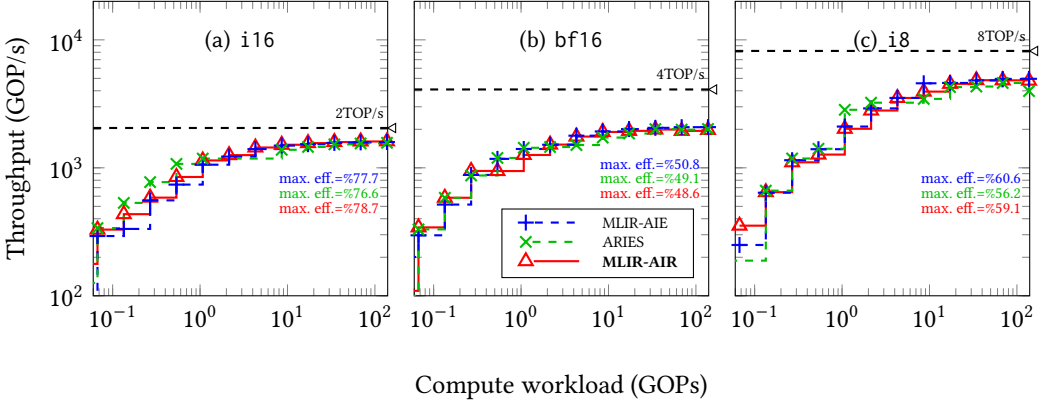


Fig. 13. Throughput versus compute workload Pareto frontiers for bfloat16, i16 and i8 matrix multiplications, with shapes swept up to $M = N = K = 4k$, for AIE tile herd sized 4×4 . Output data width was kept consistent at 16 bits for all tests—bfloat16 outputs for bfloat16 inputs, and i16 outputs for i8 and i16 inputs, respectively. The dotted line marks the theoretical peak compute throughput⁵ achievable for each herd of AIE cores at the specified precision.

The three subplots (a–c) compare performance as the `air.herd` dimensions increase. The `air.herd` dimensions are easily tunable in source code via tiling factors (Section 7.1). The peak throughput achieved scales proportionately with the tile count, demonstrating that MLIR-AIR is able to leverage increased spatial compute automatically, analyzed from the structured parallelism in the input program.

Larger problem sizes increase the computational intensity (OPs per memory access), which lead to better utilization of compute tiles in the dataflow pipeline. As we sweep across increasing problem sizes (larger K values), the throughput consistently improves. Higher K reduces the start-up effect of the dataflow pipeline by reducing the frequency of flushes as the scheduler refills accumulators with zeros, leading to increased overall performance. This trend reflects AIR’s ability to schedule larger compute tiles effectively, reducing the relative overhead of data transfers and synchronization.

Across all configurations, throughput is lower at smaller workloads (left side of each plot) due to underutilization of compute resources and startup latency at runtime. As the compute workload increases, throughput rises and asymptotically approaches the device peak. This indicates the transition from memory-bound throughput at small sizes to compute-bound throughput at larger sizes. The shape of the performance curve confirms that MLIR-AIR introduces minimal runtime overhead and supports efficient scaling into the compute-bound region. MLIR-AIR achieves up to 48.6% of peak on the 4TOP/s herd, 65.0% of peak on the 2TOP/s herd, and 48.2% of peak on the 1TOP/s herd.

To evaluate the QoR achievable by MLIR-AIR, we benchmark the performance of matrix multiplication workloads across three common AI data types supported by AMD NPU’s vector engine: i16, bf16, and i8. MLIR-AIE’s hand-optimized implementations serve as baselines. Figure 13 shows that in all three precision settings, MLIR-AIR’s compiler-generated designs achieve throughput closely tracking the Pareto frontier established by the manually optimized designs written in MLIR-AIE. This confirms MLIR-AIR’s effectiveness in generating near-optimal performance without requiring handcrafted code.

Designs generated by MLIR-AIR achieve 78.7%, 48.6% and 59.1% maximum efficiencies against the theoretical peak throughput for i16, bf16 and i8, respectively, which fall within 5 pp from the MLIR-AIE hand-optimized designs.

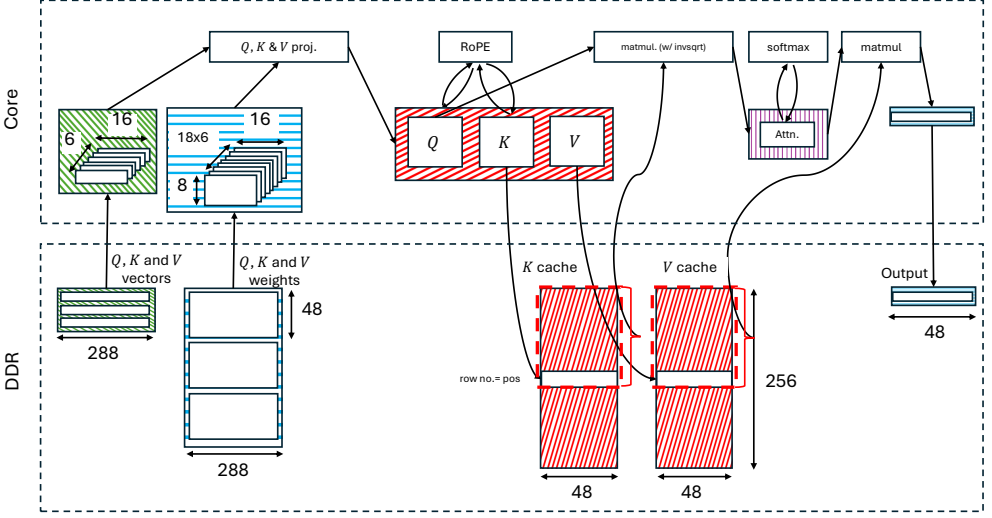


Fig. 14. An overview of the fused LLaMA 2 MHA schedule. The dataflow through memory tile is omitted for simplicity.

We also compare MLIR-AIR against ARIES, which demonstrates strong performance on smaller GEMM shapes, particularly for `i16` and `i8`. However, ARIES exhibits lower peak throughputs on these two data formats when saturated, indicating trade-offs between early-stage performance and scalability. These results further underscore MLIR-AIR’s ability to combine generality, analyzability, and performance within a unified spatial compilation flow.

9.3 Kernel Merging: LLaMA 2 Multi-Head Attention

To evaluate MLIR-AIR’s ability to express and optimize fused AI kernels, we implement a prototype of the LLaMA 2 MHA [44] block on an AMD NPU using a single AIE core. The model uses a head size of 48 and a sequence length of 256, with 6 heads multiplexed in time.

The MHA block includes projection (Q, K, V), rotary positional encoding (RoPE), softmax, and two matrix multiplications, separated by key-value (KV) caching [43]. Each operation is implemented as a generic function, composed using structured `scf.for` and `scf.parallel` loop nests. MLIR-AIR compiles this into a hardware-aware schedule, by mapping operations to NPU components such as DMA channels, BDs, and NPU compute tiles.

KV caching is implemented as loop-nested `air.channel` operations targeting persistent DDR memories, holding a cache size of 48×256 data for K and V , respectively. DMA channel and BD reuse opportunities are captured via `air.channel` merging described previously in Section 7.4.2. Correctness in placement and buffer management is enforced via MLIR-AIR’s dependency analysis, which generates proper synchronization via `air.token` values.

Table 3 profiles the end-to-end latency of each head, implemented with kernels dispatched both individually and fused together, with host and runtime dispatch overheads included. When each component executes as an independent kernel, total latency is $834\mu s$. With all kernels fused as one, latency is reduced to $373\mu s$ —achieving a $2.24\times$ speedup by eliminating dispatch overheads, amortizing reconfiguration cost, and leveraging data locality within the NPU tile’s local memory.

While this prototype does not yet exploit spatial parallelism across multiple AIE cores, it highlights MLIR-AIR’s ability to concisely represent and optimize non-trivial transformer blocks. The

Table 3. LLaMA2 MHA evaluation.

Component	Lines of code	Latency (μ s)	Speedup
Q, K and V vector projection	16	169	–
RoPE	37	121	–
matmul (w/ invsqrt)	39	283	–
softmax	26	115	–
matmul	37	146	–
Total	155	834	–
Fused	155	373	2.24×

full implementation is written in 155 lines of high-level MLIR, demonstrating the expressiveness of AIR abstractions for modeling modern AI computation and communication patterns.

10 Future Directions

We identify three key directions for extending MLIR-AIR’s applicability and automation in spatial compiler workflows:

Multi-Target Hardware Support. MLIR-AIR envisages support for multiple hardware platforms beyond NPUs, including compilation to GPUs using long-running persistent kernels and integration with user-developed accelerators implemented in FPGAs. This requires extending our backend abstractions and lowering pipelines to target architecture-specific runtimes, scheduling models and memory hierarchies.

Support for Heterogeneous Runtime Coordination. As modern systems increasingly include CPUs, GPUs, and NPUs on a shared die, MLIR-AIR can be a common abstraction supporting runtime coordination across heterogeneous devices. This includes lowering AIR to multiple backend runtimes (e.g., ROCr, XRT) and managing inter-accelerator data movement and synchronization.

Cross-Device Launch Semantics. Some MLIR-AIR features such as data movement over explicit channels, and resource management using segments may have applicability in scaling beyond a single device, we plan to explore how an appropriate runtime might use `air.launch` to enable multi-device dispatch. This includes scaling launches dynamically across available hardware based on runtime resource availability, allowing for coordinated execution across multiple accelerators on one host, and multiple hosts within a compute cluster.

11 Conclusion

MLIR-AIR introduces a structured, extensible compiler abstraction for mapping high-level AI programs onto spatial architectures. By providing explicit constructs for asynchronous parallelism, data movement, and compute scheduling, AIR enables platform-agnostic, analyzable code generation without sacrificing performance. Our extensive evaluation demonstrates that AIR provides both high expressiveness and efficiency, while maintaining a low abstraction overhead. We believe MLIR-AIR provides a strong foundation for future spatial compiler infrastructures.

References

- [1] Dennis Abts, John Kim, Garrin Kimmell, Matthew Boyd, et al. 2022. The Groq Software-defined Scale-out Tensor Streaming Multiprocessor : From chips-to-systems architectural overview. In *IEEE Hot Chips 34 Symposium (HCS)*.
- [2] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, et al. 2022. An MLIR-based Compiler Flow for System-level Design and Hardware Acceleration. In *IEEE/ACM International Conference on Computer-Aided Design*.
- [3] The Chromium Authors. 2025. *Perfetto*. <https://perfetto.dev/docs/>
- [4] The IREE Authors. 2019. *IREE*. <https://iree.dev/>
- [5] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, et al. 2008. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *International Conference on Supercomputing*.
- [6] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model is More Widely Applicable than You Think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*.

- [7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
- [8] Cerebras. 2025. *Cerebras Wafer Scale Engine*. <https://www.cerebras.ai/chip>
- [9] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.
- [10] Prasanth Chatarasi, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. 2021. Marvel: A Data-centric Approach for Mapping Deep Learning Operators on Spatial Accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 1 (2021), 1–26.
- [11] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.
- [12] Microsoft Corporation. 2025. *Triton-shared*. <https://github.com/microsoft/triton-shared>
- [13] Advanced Micro Devices. 2025. *AI Engine*. <https://www.amd.com/en/products/adaptive-socs-and-fpgas/technologies/ai-engine.html>
- [14] Advanced Micro Devices. 2025. *MLIR-AIE*. <https://xilinx.github.io/mlir-aie/>
- [15] Advanced Micro Devices. 2025. *ROCm*. <https://rocm.docs.amd.com/projects/ROCm-Runtime/en/latest/>
- [16] Advanced Micro Devices. 2025. *Ryzen 7 7840U*. <https://www.amd.com/en/products/processors/laptop/ryzen/7000-series/amd-ryzen-7-7840u.html>
- [17] Advanced Micro Devices. 2025. *XRT*. <https://xilinx.github.io/XRT/master/html/index.html>
- [18] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*.
- [19] T Hariprasad, G Vidhyakaran, K Seenu, and Chandrasegar Thirumalai. 2017. Software Complexity Analysis using Halstead Metrics. In *International Conference on Trends in Electronics and Informatics (ICEI)*.
- [20] Erika Hunhoff, Joseph Melber, Kristof Denolf, Andra Bisca, Samuel Bayliss, Stephen Neuendorffer, Jeff Fifield, Jack Lo, Pranathi Vasireddy, Phil James-Roxby, and Eric Keller. 2025. Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.
- [21] INTEL. 2025. *Intel NPU*. <https://edc.intel.com/content/www/us/en/design/products/platforms/details/arrow-lake-s/core-ultra-200s-series-processors-datasheet-volume-1-of-2/intel-neural-processing-unit-intel-npu/>
- [22] Geonhwa Jeong, Gokcen Kestor, Prasanth Chatarasi, Angshuman Parashar, Po-An Tsai, Sivasankaran Rajamanickam, Roberto Gioiosa, and Tushar Krishna. 2021. Union: A Unified HW-SW Co-design Ecosystem in MLIR for Evaluating Tensor Operations on Spatial Accelerators. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
- [23] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, et al. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Annual International Symposium on Computer Architecture*.
- [24] Michele Lacchia. 2025. *Radon*. <https://github.com/rubik/radon/tree/master>
- [25] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.
- [26] Chris Lattner and Jacques Pienaar. 2019. MLIR Primer: A Compiler Infrastructure for the End of Moore’s Law.
- [27] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
- [28] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2022. COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry. In *Languages and Compilers for Parallel Computing*.
- [29] NVIDIA. 2020. CUDA, release: 10.2.89. <https://developer.nvidia.com/cuda-toolkit>
- [30] OpenAI. 2025. *TRITON*. <https://triton-lang.org/main/index.html>
- [31] Raghu Prabhakar, Ram Sivaramakrishnan, Darshan Gandhi, Yun Du, et al. 2024. SambaNova SN40L: Scaling the AI Memory Wall with Dataflow and Composition of Experts. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
- [32] Qualcomm. 2025. *Qualcomm AI Engine*. <https://www.qualcomm.com/products/technology/processors/ai-engine>
- [33] Alejandro Rico, Satyaprakash Pareek, Javier Cabezas, David Clarke, et al. 2024. AMD XDNA™ NPU in Ryzen™ AI Processors. *IEEE Micro* 44, 6 (2024), 73–82.
- [34] André Rösti and Michael Franz. 2025. Unlocking the AMD Neural Processing Unit for ML Training on the Client Using Bare-Metal-Programming Tools. In *IEEE International Symposium on Field-Programmable Custom Computing*

Machines (FCCM).

- [35] Alexander C Rucker, Shiv Sundram, Coleman Smith, Matthew Vilim, Raghu Prabhakar, Fredrik Kjølstad, and Kunle Olukotun. 2024. Revet: A Language and Compiler for Dataflow Threads. In *International Symposium on High-Performance Computer Architecture (HPCA)*.
- [36] Amit Sabne. 2020. XLA: Compiling Machine Learning for Peak Performance. <https://research.google/pubs/xla-compiling-machine-learning-for-peak-performance/>.
- [37] Gagandeep Singh. 2022. Designing, modeling, and optimizing data-intensive computing systems. *arXiv preprint arXiv:2208.08886* (2022).
- [38] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. 2021. FPGA-based near-memory acceleration of modern data-intensive applications. *IEEE Micro* 41, 4 (2021), 39–48.
- [39] Gagandeep Singh, Mohammed Alser, Kristof Denolf, Can Firtina, Alireza Khodamoradi, Meryem Banu Cavlak, Henk Corporaal, and Onur Mutlu. 2024. RUBICON: a framework for designing efficient deep learning-based genomic basecallers. *Genome Biology* 25, 1 (2024), 49.
- [40] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gomez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. 2020. NERO: A near high-bandwidth memory stencil accelerator for weather prediction modeling. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 9–17.
- [41] Gagandeep Singh, Alireza Khodamoradi, Kristof Denolf, Jack Lo, Juan Gomez-Luna, Joseph Melber, Andra Bisca, Henk Corporaal, and Onur Mutlu. 2023. SPARTA: Spatial Acceleration for Efficient and Scalable Horizontal Diffusion Weather Stencil Computation. In *Proceedings of the 37th International Conference on Supercomputing*. 463–476.
- [42] Gagandeep Singha, Dionysios Diamantopoulosb, Juan Gómez-Lunaa, Sander Stuijkc, Henk Corporaalc, and Onur Mutlua. 2022. LEAPER: Fast and accurate FPGA-based system performance prediction via transfer learning. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 499–508.
- [43] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. RoFormer: Enhanced Transformer with Rotary Position Embedding. *Neurocomput.* 568, C (2024).
- [44] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv:2307.09288 [cs.CL]* <https://arxiv.org/abs/2307.09288>
- [45] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.
- [46] Hanchen Ye, Hyegang Jun, and Deming Chen. 2024. HIDA: A Hierarchical Dataflow Compiler for High-level Synthesis. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [47] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: Enabling Automatic Mapping for Tensor Computations on Spatial Accelerators with Hardware Abstraction. In *Annual International Symposium on Computer Architecture*.
- [48] 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 *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*.

A Input IR to MLIR-AIR's Vector-add Example

Listing 7. Element-wise vector add, described in upstream MLIR dialects.

```

1 func.func @eltwise_add(%arg0: memref<65536xf32>, %arg1: memref<65536xf32>, %arg2: memref<65536xf32>) {
2   %c65536 = arith.constant 65536 : index
3   %c2048 = arith.constant 2048 : index
4   %c1024 = arith.constant 1024 : index
5   %c1 = arith.constant 1 : index
6   %c2 = arith.constant 2 : index
7   %c0 = arith.constant 0 : index
8   scf.parallel (%arg3) = (%c0) to (%c2) step (%c1) {
9     %alloc = memref.alloc() : memref<1024xf32, 2>
10    %alloc_0 = memref.alloc() : memref<1024xf32, 2>
11    %alloc_1 = memref.alloc() : memref<1024xf32, 2>
12    scf.for %arg4 = %c0 to %c65536 step %c2048 {
13      %subview = memref.subview %arg0[0] [1024] [1] : memref<65536xf32> to memref<1024xf32>
14      memref.copy %subview, %alloc : memref<1024xf32> to memref<1024xf32, 2>
15      %subview_2 = memref.subview %arg1[0] [1024] [1] : memref<65536xf32> to memref<1024xf32>
16      memref.copy %subview_2, %alloc_0 : memref<1024xf32> to memref<1024xf32, 2>
17      scf.for %arg5 = %c0 to %c1024 step %c1 {
18        %0 = memref.load %alloc[%arg5] : memref<1024xf32, 2>
19        %1 = memref.load %alloc_0[%arg5] : memref<1024xf32, 2>
20        %2 = arith.addf %0, %1 : f32
21        memref.store %2, %alloc_1[%arg5] : memref<1024xf32, 2>
22      }
23      %subview_3 = memref.subview %arg2[0] [1024] [1] : memref<65536xf32> to memref<1024xf32>
24      memref.copy %alloc_1, %subview_3 : memref<1024xf32, 2> to memref<1024xf32>
25      memref.dealloc %alloc : memref<1024xf32, 2>
26      memref.dealloc %alloc_0 : memref<1024xf32, 2>
27      memref.dealloc %alloc_1 : memref<1024xf32, 2>
28    }
29    scf.reduce
30  }
31  return
32 }
```

B AIR Python Bindings to MLIR-AIR's Vector-add Example

Listing 8. Element-wise vector add, described in AIR's Python bindings.

```
@module_builder
def build_module(n, tile_n, np_dtype_in):
    a_size = [n]
    b_size = a_size
    out_size = a_size
    xrt_dtype_in = type_mapper(np_dtype_in)
    num_tiles = 2
    assert n % (tile_n * num_tiles) == 0
    # L3 MemRefTypes
    l3memrefTy = MemRefType.get(a_size, xrt_dtype_in)
    # L1 MemRefTypes
    l1MemrefTy = MemRefType.get(
        shape=[tile_n],
        element_type=xrt_dtype_in,
        memory_space=IntegerAttr.get(T.i32(), MemorySpace.L1),
    )
    @FuncOp.from_py_func(l3memrefTy, l3memrefTy, l3memrefTy)
    def eltwise_add(arg0, arg1, arg2):
        @herd(
            name="herd_0",
            sizes=[1, num_tiles],
            operands=[arg0, arg1, arg2],
        )
        def herd_body(_tx, _ty, _sx, _sy, _l3_a, _l3_b, _l3_c):
            l1_a_data = AllocOp(l1MemrefTy, [], [])
            l1_b_data = AllocOp(l1MemrefTy, [], [])
            l1_out_data = AllocOp(l1MemrefTy, [], [])
            for _l1_ixv in range(0, n, tile_n * num_tiles):
                offset_map = AffineMap.get(0, 2,
                    [
                        AffineExpr.get_add(
                            AffineSymbolExpr.get(0),
                            AffineExpr.get_mul(
                                AffineSymbolExpr.get(1),
                                AffineConstantExpr.get(tile_n),
                            ),
                        ),
                    ],
                )
                offset = affine_apply(offset_map, [_l1_ixv, _ty])
                dma_memcpy_nd(l1_a_data, _l3_a,
                    src_offsets=[offset],
                    src_sizes=[tile_n],
                    src_strides=[1],
                )
                dma_memcpy_nd(l1_b_data, _l3_b,
                    src_offsets=[offset],
                    src_sizes=[tile_n],
                    src_strides=[1],
                )
                for i in range(tile_n):
                    val_a = load(l1_a_data, [i])
                    val_b = load(l1_b_data, [i])
                    val_out = arith.addf(val_a, val_b)
                    store(val_out, l1_out_data, [i])
                yield_([[]])
            dma_memcpy_nd(_l3_c, l1_out_data,
                dst_offsets=[offset],
                dst_sizes=[tile_n],
                dst_strides=[1],
            )
            DeallocOp(l1_a_data)
            DeallocOp(l1_b_data)
            DeallocOp(l1_out_data)
            yield_([[]])
```