AIEHalide: Compiling Halide to Spatial NPU Dataflow with Constrained Autoscheduling

Abnikant Singh
IIIT Hyderabad
Hyderabad, India
Advanced Micro Devices, Inc.
Bengaluru, India
abnikant.singh@research.iiit.ac.in

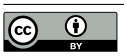
Suresh Purini
IIIT Hyderabad
Hyderabad, India
suresh.purini@iiit.ac.in

Abstract

AMD XDNA™ neural processing units (NPUs) are integrated into general-purpose processors such as Ryzen™ AI, delivering high performance-per-watt for machine-learning and image-processing pipelines. The architecture is a spatial array of tiles, each hosting a VLIW processor with SIMD execution units and 64 KB of scratchpad, interconnected by a programmable network for explicit dataflow. This explicit dataflow can deliver high throughput, but achieving it requires careful programming: computation distributed across tiles, data movement staged through a three-level scratchpad hierarchy, and intra-tile SIMD pipelines fully utilized—all under hard scratchpad-capacity constraints. IRON [9] provides a close-to-metal Python programming model with full control over dataflow topology, DMA descriptors, and tile placement. A complementary high-level path is missing: one that maps programs to the spatial dataflow architecture, automates tile placement and data plumbing, and ports cleanly across hardware generations.

We present AIEHalide, to our knowledge the first compiler that maps Halide programs onto spatial dataflow NPUs. Halide programs map naturally onto NPU dataflow: pipeline stages become compute tiles, and producer–consumer edges become streaming channels. Scheduling directives control tile sizes, core mappings, and data layouts, and the algorithm–schedule separation provides cross-generation portability. The compiler derives DMA descriptors, buffer depths, and host sequences automatically, emitting MLIR lowered through the mlir-aie toolchain. An autoscheduler formulates NPU scheduling as a feasibility-first search over tile geometries, buffer depths, and data layouts under hard, compile-or-fail scratchpad and channel constraints, ranked by an analytical cost model.

On two NPU generations the same Halide source achieves int8 GEMM throughput of 5.42 TOPS on XDNA (53% of peak) and 25.46 TOPS on XDNA 2 (62% of peak), reducing user code by 2.8×. The autoscheduler discovers asymmetric tile shapes and platform-specific layouts without manual tuning. Beyond GEMM, 43 operators—including multi-stage stencil and normalization pipelines—compile and pass verification on both generations with no source changes; deeply fused ML pipelines remain future work.



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846838

CCS Concepts

• **Software and its engineering** → **Compilers**; **Domain specific languages**; *Source code generation*; • **Computer systems organization** → *Parallel architectures*; *Heterogeneous (hybrid) systems.*

Keywords

Spatial NPUs, Halide, autoscheduler, MLIR, compilers

1 Introduction

Spatial dataflow accelerators are reshaping on-device inference, due to their performance-per-watt advantage over CPUs and GPUs. AMD XDNA™ neural processing units (NPUs) pack a 2D array of AI Engine (AIE) tiles with Very Long Instruction Word (VLIW) compute cores, connected via a network-on-chip (NoC) over a three-level memory hierarchy. The first generation, XDNA, has 20 tiles delivering 10 TOPS; the second, XDNA 2, has 32 tiles reaching 50 TOPS [18]. They are integrated into SoCs such as Ryzen™ AI, targeting edge and consumer PCs.

Unlike CPUs and GPUs, where hardware caches absorb locality mistakes at the cost of performance, AIE tiles impose *hard* scratchpad limits—exceeding L1 (per-tile private data memory, 64 KB) or L2 (shared MemTile memory, 512 KB per column) prevents compilation. All data movement across the memory hierarchy is software-managed through DMA descriptors and streaming channels [3, 9]. Three challenges make productive NPU programming difficult today.

Explicit dataflow programming requires expert effort. Current NPU programming models interleave the algorithm with detailed hardware decisions. IRON [9], the primary close-to-metal AIE framework, gives experts full control over channel topology, buffer depths, DMA descriptors, and tile placement—but a 4×4 GEMM takes ∼250 lines of infrastructure alongside the compute, and a two-stage blur pipeline ∼180. Higher-level abstractions such as ARIES [24] reduce verbosity but still embed tile sizes, layouts, and microkernel shapes in the algorithm description. Performance-critical decisions propagate across the entire design, making iterative optimization costly and error-prone.

Cross-generation portability remains a challenge. When hardware evolves—from XDNA to XDNA 2—microkernel shapes change (`mmul<4,8,4>` vs. `mmul<4,8,8>` for bf16), register layouts differ, and accumulator widths grow (512-bit on AIE2 vs. 2048-bit on AIE2P).

AMD, the AMD Arrow logo, AMD Ryzen, AMD XDNA, Versal, Xilinx, and combinations thereof are trademarks of Advanced Micro Devices, Inc.

Because existing frameworks embed these parameters directly in the program, every design is rewritten and re-tuned per generation; the effort compounds across tens of operators.

Performance optimization spans a large parameter space. High utilization requires jointly selecting tile sizes, buffer depths, layouts, L2 staging factors, and microkernel shapes—a large parameter space whose axes interact in non-obvious ways. Recent work on AMD NPUs demonstrates that even single-axis tuning matters substantially: reduction-dimension staging and input layout alone yield large gains [18]; asymmetric tile buffering independently improves arithmetic intensity [19]. Both rely on manual analysis and expert profiling. No existing AMD NPU autoscheduler explores the joint space.

We present **AIEHalide**, a compiler that maps Halide [16] programs onto AMD NPUs. Halide's programming model maps directly onto a spatial dataflow array: `Funcs` linked by producer–consumer edges become compute tiles joined by streaming channels, and fusion becomes tile-local storage. AIEHalide takes this `Func` graph as the array's programming model, lowering it through mlir-aie [3] to hardware dataflow. The core technical challenge is twofold: *reconstructing* pipeline structure from Halide's lowered IR—where stage boundaries and dependence edges have been flattened—and mapping it onto the NPU's *physical, non-virtualized* L1/L2 scratchpads under hard, compile-or-fail capacity and channel limits.

Contributions.

(1) **Halide-to-dataflow compilation with a schedule-driven control surface.** We compile Halide's functional IR to spatial NPU dataflow: standard Halide primitives drive NPU parameters (tile sizes, reduction blocking, core distribution, DMA sequences) without altering the algorithm language. A small set of three optional NPU-specific directives (`aie_dataflow`, `aie_fuse_with`, `aie_kernel`)—none required for correctness—exposes buffer depths, L2 staging, layouts, stage fusion, and external kernel attachment. A centralized device model encodes tile grid, memory hierarchy, microkernel shapes, and DMA capabilities; updating it suffices to port programs across generations. Unlike hand-written mlir-aie, one Halide source yields both generations' dataflow, host application, and DMA sequences with a *ranked* set of buildable schedules, and reuses an existing peak microkernel unchanged via `aie_kernel` (§3, §3.1, §3.6).

(2) **Constrained NPU autoscheduler.** We formulate NPU scheduling as constrained optimization, jointly searching tile sizes, core count, microkernel shape, buffer depths, L2 staging factor, and data layout. An analytical roofline cost model ranks candidates accurately. To our knowledge, this is the first autoscheduler for AMD NPUs that searches this joint parameter space (§4).

(3) **Performance evaluation across two NPU generations from a single source.** On XDNA, int8 GEMM achieves 5.42 TOPS (53% of peak); on XDNA 2, the same source reaches 25.46 TOPS int8 (62%) and 6.18 TOPS int16 (60%)—zero source changes across generations. These numbers come from the best execution configurations the autoscheduler found per platform: asymmetric tiles and platform-specific layouts. The GEMM kernels are hand-optimized C++ attached via the `aie_kernel` directive—this work targets dataflow optimization, not kernel optimization. The compiler does support auto-generated kernels for all operator classes, though auto-generated GEMM currently trails hand-optimized by ∼15–30%. We evaluate seven GEMM dtypes, sizes 512–8192, 19 vision pipelines, and 24 ML operators on both platforms (§5).

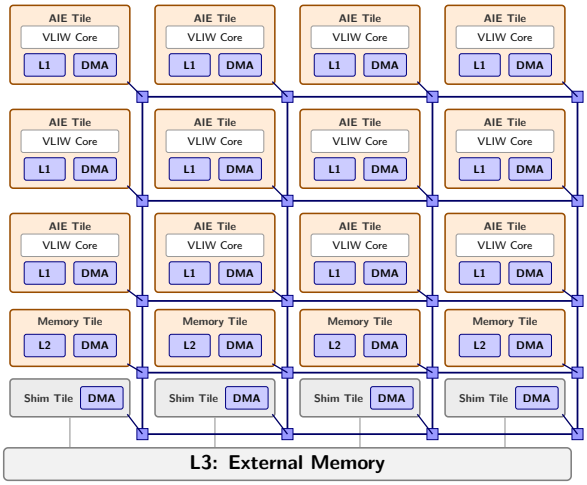


Figure 1: Simplified AMD AIE tile-array architecture. Each AIE tile contains a VLIW processor core, 64 KB L1 scratchpad, and a DMA engine. Memory tiles provide shared 512 KB L2 storage per column; shim tiles bridge the array to external DDR (L3). The figure is schematic: actual XDNA has 4×5 = 20 compute tiles; XDNA 2 has 4×8 = 32 compute tiles.

2 Background and Motivation

We introduce the AI Engine tile array and its software-managed memory hierarchy (§2.1), illustrate the explicit dataflow style used by IRON [9]—the current close-to-metal programming model (§2.2)—and show how the AIEHalide compiler automates the same dataflow via Halide's algorithm–schedule separation, while preserving expert control (§2.3).

2.1 AI Engine Tile Array

AMD Ryzen™ AI processors integrate an AI Engine (AIE) tile array as an on-chip NPU [2]. Figure 1 shows the XDNA (AIE2) layout: 5 columns of 4 compute tiles (20 cores at 1.25 GHz, 512-bit VLIW vector units), one memory tile per column (*MemTile*), and a row of *shim* DMA tiles—interface tiles that bridge the array to external DDR memory—at the boundary. Each compute tile delivers 256 int8 MACs per cycle (64 int16, 16 int32, or 8 FP32), for an aggregate peak of ∼10 TOPS (int8). The second-generation XDNA 2 (AIE2P) array doubles the array to 8 columns (32 cores) and widens each tile's datapath to 512 int8 MACs/cycle, reaching ∼50 TOPS [18].

Software-managed three-level hierarchy. AIE tiles have *no* hardware caches. All data movement is explicit and staged through three levels: **L1** (64 KB private scratchpad per compute tile), **L2** (512 KB per MemTile, shared across the column's compute tiles), and **L3** (external double data rate (DDR) memory, via shim DMAs). L1 is a

hard capacity constraint: exceeding 64 KB prevents compilation. Every inter-level transfer is programmed as a DMA buffer descriptor (BD) with multi-dimensional sizes and strides (3D on compute/shim tiles, 4D on memory tiles), and every inter-tile exchange travels over a configurable streaming channel synchronized by hardware locks. Because each BD dimension has its own size and stride, a DMA can reorder the data it streams, so an operand can arrive in a different layout than the one it had in memory. §3.3 builds on this to perform layout reformats during the transfer itself, at no compute-side cost.

ObjectFIFOs. The `mlir-aie` [3] toolchain exposes streaming channels via `aie.objectfifo`: a typed, configurable-depth queue in which producers acquire/write/release slots and consumers acquire/read/release them. A single ObjectFIFO may broadcast to several consumers. ObjectFIFOs are the lowest-level data-movement primitive an IRON programmer works with: every pipeline edge, every L3→L2→L1 staging step, and every inter-tile broadcast is expressed by hand as one or more of them. Listing 1 in the next subsection shows ObjectFIFOs in a concrete IRON design.

2.2 Explicit Dataflow with IRON

To illustrate the explicit dataflow style, consider a two-stage pipeline: **stage 1** computes a three-point horizontal blur $b(x, y) = (\mathrm{in}(x-1, y) + \mathrm{in}(x, y) + \mathrm{in}(x+1, y))/3$, and **stage 2** computes an unsharp sharpen $s(x, y) = \mathrm{clamp}(2 \cdot \mathrm{in}(x, y) - b(x, y), 0, 255)$. The sharpen stage reads *both* the blurred output and the original input—a canonical producer–consumer dataflow with a broadcast.

Listing 1 shows the design in IRON's Python API [9] (Worker/Runtime/Program style). IRON gives the programmer full control over the dataflow, essential for tuning expert kernels. The design makes several hardware decisions explicit:

(1) **Data channels.** Three ObjectFIFOs carry data between stages, each with an explicit element type, tile shape, and buffer depth.
(2) **Broadcast topology.** The input is consumed by both blur and sharpen; two consumer handles (`f_in.cons()`) express this broadcast to the DMA engine.
(3) **Acquire/release protocol.** Each core function acquires input and output slots, computes, and releases them in matching order—the protocol that drives the streaming pipeline.
(4) **DMA descriptors.** The runtime sequence programs shim DMA `sizes/strides` that must agree with the ObjectFIFO element shapes and the L3 image layout.
(5) **Worker–handle wiring.** Each worker is constructed from a core function and an explicit list of ObjectFIFO endpoints; a placer automatically maps workers to physical tiles.

This explicit style delivers fine-grained control valuable for peak-performance tuning. The trade-off is that the algorithm—the two inner-loop bodies tagged `ALG` in Listing 1—is interleaved with data-movement logic, and performance-critical decisions (tile sizes, layout, channel topology) propagate across the entire listing; in larger pipelines, additional decisions (L2 staging, microkernel selection) compound the problem. AIEHalide automates these infrastructure decisions from a higher-level description while preserving expert override where needed.

Listing 1: Two-stage blur → sharpen pipeline in IRON. Algorithm occupies only the inner loop bodies (tagged ALG); everything else is data-movement plumbing.

```
from aie.iron import ObjectFifo, Worker, Runtime, Program
from aie.iron.placers import SequentialPlacer
from aie.iron.device import NPU1Col1
import numpy as np
H, W = 4, 64
tty = np.ndarray[(H, W),    np.dtype[np.int16]]
mty = np.ndarray[(H, 16*W), np.dtype[np.int16]]
# Data channels
f_in  = ObjectFifo(tty, depth=2)
f_mid = ObjectFifo(tty, depth=2)
f_out = ObjectFifo(tty, depth=2)
# 1->many broadcast for sharpen
in_for_blur    = f_in.cons()
in_for_sharpen = f_in.cons()
# Stage 1: horizontal blur
def blur_fn(of_in, of_out):
    a = of_in.acquire(1)
    b = of_out.acquire(1)
    for i in range_(H):              # ALG
        for j in range_(1, W-1):     # ALG
            b[i,j] = (a[i,j-1]+a[i,j]+a[i,j+1])//3
    of_in.release(1); of_out.release(1)
blur_w = Worker(blur_fn,
    fn_args=[in_for_blur, f_mid.prod()])
# Stage 2: unsharp
def sharp_fn(of_orig, of_blur, of_out):
    o = of_orig.acquire(1)
    b = of_blur.acquire(1)
    c = of_out.acquire(1)
    for i in range_(H):              # ALG
        for j in range_(W):          # ALG
            v = 2*o[i,j] - b[i,j]
            c[i,j] = max(0, min(255, v))
    of_orig.release(1); of_blur.release(1)
    of_out.release(1)
sharp_w = Worker(sharp_fn,
    fn_args=[in_for_sharpen, f_mid.cons(), f_out.prod()])
# Runtime: host DMA + worker orchestration
rt = Runtime()
with rt.sequence(mty, mty) as (in_buf, out_buf):
    rt.start(blur_w, sharp_w)
    rt.fill (f_in.prod(),  in_buf,
             sizes=[1,1,H,16*W], strides=[0,0,16*W,1])
    rt.drain(f_out.cons(), out_buf,
             sizes=[1,1,H,16*W], strides=[0,0,16*W,1],
             wait=True)
print(Program(NPU1Col1(), rt).resolve_program(
    SequentialPlacer()))
```

Listing 2: Blur → sharpen pipeline in Halide: 4 algorithm lines + 2 schedule lines.

```
Func input, blur, sharpen;
input(x, y)   = in_buf(x, y);
blur(x, y)    = (input(x-1,y)+input(x,y)+input(x+1,y))/3;
sharpen(x, y) = clamp(2*input(x,y)-blur(x,y), 0, 255);

blur.compute_root();    // stage 1 AIE core
sharpen.compute_root(); // stage 2 AIE core
```

2.3 The Same Pipeline in Halide

Halide [16] separates the *algorithm*—what to compute—from the *schedule*—how and where to compute it. Listing 2 expresses the same blur→sharpen pipeline.

The algorithm is 4 lines: three `Func`s with producer–consumer edges expressed by name references (`sharpen` reads `blur` and `input`). The schedule is 2 lines: `compute_root()` marks each stage as a distinct pipeline node. The compiler derives ObjectFIFOs, acquire/release sequences, DMA descriptors, and the runtime sequence automatically; the broadcast of `input` is implicit in the `Func` reference graph and reconstructed from the IR (§3.2).

Listing 3: Halide GEMM: 3 algorithm lines + 4 schedule lines.

```
// Algorithm: C = A * B  (bf16 -> f32)
RDom r(0, K);
C(i, j) = 0.0f;
C(i, j) += cast<float>(A(i,r))*cast<float>(B(r,j));

// Schedule: standard Halide
C.tile(i, j, io, jo, ii, ji, 64, 32);  // spatial
C.split(r.x, ro, ri, 64);              // reduction
C.reorder(ri, ii, ji, ro, io, jo);     // loop order
C.vectorize(ji).vectorize(ii);         // vectorize
```

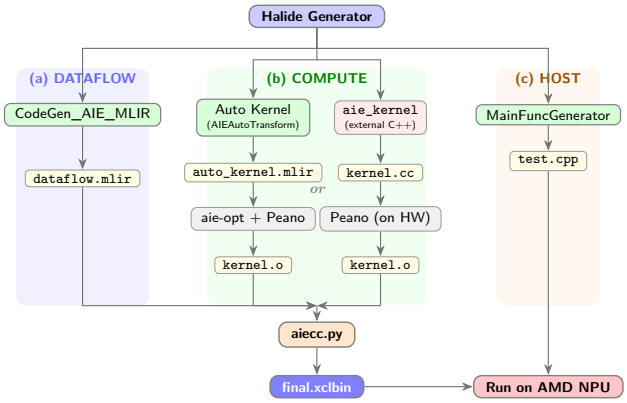


Figure 2: AIEHalide compilation flow. Three concurrent paths converge at `aiecc.py`: (a) a dataflow path emitting AIE/AIEX MLIR with `link_with` references to kernel objects; (b) a kernel path that supplies kernels either auto-generated (via `aie-opt` + Peano [4]) or as external hand-optimized C++ via `aie_kernel`; and (c) a host path emitting an XRT application. Together they produce an `xclbin` binary.

Scheduling is the control surface, not the algorithm. The algorithm is identical to a CPU- or GPU-targeted Halide program; only the schedule changes, mapping standard Halide primitives onto NPU parameters. Listing 3 shows this on GEMM: from 3 algorithm lines and 4 schedule lines the compiler (§3) derives the L1 tile (T_m=64, T_k=64, T_n=32), the core distribution over the 4×4 grid, the ObjectFIFO topology, and every DMA descriptor. An IRON implementation of this kernel is ∼250 lines of Python; the Halide source is 7 lines that compile to the same dataflow. The remainder of the paper explains how the compiler bridges the two representations and how the constrained autoscheduler (§4) searches the schedule space.

3 Compiler Architecture

A Halide pipeline implicitly describes a dataflow graph: each `Func` is a compute node, inter-`Func` references are producer–consumer edges, and scheduling directives determine tile sizes and loop ordering. Mapping this onto a spatial NPU is conceptually direct—`Funcs` become AIE compute tiles, dataflow edges become streaming channels—but Halide's lowering flattens the structure into nested loops, dissolving stage boundaries and removing explicit dependence edges. AIEHalide's core task is to recover this structure from the lowered intermediate representation (IR) and emit a complete NPU application—dataflow, kernels, and host code—from a single Halide generator. The algorithm language is unchanged from Halide; standard scheduling primitives drive the default flow, with a small set of optional NPU-specific directives available for performance tuning (Table 1).

Three concurrent compilation paths produce the artifacts (Figure 2): (a) a *dataflow path* that recovers schedule and DAG from lowered IR and emits the AIE/AIEX dialect [3] of MLIR [11]—tile placement, streaming channels, DMA descriptors, and runtime sequences; (b) a *kernel path* that produces each core's compute kernel—by default lowering the Halide compute expressions through the linalg, transform, and AIEVec dialects [3], or attaching a hand-tuned external microkernel when a `Func` declares `aie_kernel()` (§3.4); and (c) a *host path* that emits a Xilinx™ Runtime (XRT) application for buffer management and NPU invocation. The dataflow MLIR, the per-core ELF that Peano [4] produces (either from the AIEVec lowering or from compiling an external C++ kernel), and the host application all flow into `aiecc.py`, which assembles the result into an `xclbin` binary.

3.1 Scheduling API and Dataflow Control

The compiler's interface to the user is the standard Halide scheduling language. The algorithm is written as Halide `Funcs`; the schedule is expressed with Halide's existing primitives (Table 1). The compiler interprets these against the lowered IR (via the `analyze_schedule_from_ir` pass; §3.2) and derives all NPU-specific parameters automatically. The same schedule that produces a correct CPU binary also produces a correct NPU binary; no AIE-specific syntax is required for default behavior.

Table 1 lists each primitive and the NPU parameter it drives. `tile(i, j,` T_m`,` T_n`)` sets the L1 tile dimensions; `split(r, ro, ri,` T_k`)` on a reduction dimension yields the K-reduction tile and `mmul` blocking; `parallel` on a spatial loop sets the core count and tile distribution; `reorder` controls loop nesting and DMA access patterns; `vectorize` triggers vector lowering through `aievec.matmul`; `compute_root` marks a `Func` as its own dataflow worker (joined to others by streaming channels), while `compute_at` nests a producer into its consumer's stage.

For performance tuning, three optional NPU-specific directives are available. `aie_dataflow()` attaches per-channel buffer depths, input layout (row- or column-major), and the L2 staging factor σ to a `Func`; defaults (double-buffered inputs, single-buffered output, row-major, σ=1) apply otherwise, and the mechanics are described in §3.3. `aie_fuse_with(g)` attaches a sibling `Func` to the same AIE tile, merging the two stages. This is useful when two or more kernels are individually compute-light: fusing them onto one tile removes the inter-tile DMA between them. `aie_kernel(path)` attaches a hand-optimized C++ microkernel via the emitted MLIR's `link_with` attribute. The autoscheduler (§4) emits all three through the same API.

3.2 Recovering from Lowered IR

Halide's lowering flattens the producer–consumer structure into nested loops and buffer allocations, dissolving the stage boundaries, dependence edges, *and* the schedule parameters a spatial mapping needs. AIEHalide intercepts that lowered function once and recovers both in the two phases of Algorithm 1.[1] InferSchedule walks

[1]One-to-one with `CodeGen_AIE_MLIR::compile`: InferSchedule is `analyze_schedule_from_ir`; ReconstructDAG is `extract_from_lowered_func`

Table 1: Halide schedule directives and the AIE parameters they drive.

Halide directive	Derived AIE parameter
`.tile(i,j,`T_m`,`T_n`)`	L1 tile dims; channel buffer sizes
`.split(r,ro,ri,`T_k`)`	K-reduction tile; `mmul` blocking
`.vectorize(loop)`	Vector ops; `aievec.matmul`
`.parallel(loop)`	Core count $C_{\max}$; tile distribution
`.reorder()`	Loop nesting; DMA access patterns
`.compute_root()`	Stage boundary: own worker (`.compute_at` nests into consumer)
Optional (expert tuning):	
`aie_dataflow()`	buffer depths, L2 staging factor, data layout
`aie_fuse_with(f)`	Stage fusion: co-locate on one tile (removes inter-tile DMA)
`aie_kernel(path)`	Hand-optimized kernel object

Algorithm 1: RecoverFromLoweredIR

Input: lowered Halide function F (after bounds inference; buffer accesses kept multi-dimensional)
Output: schedule Θ, nodes V, edges E

```
// Phase 1: InferSchedule
Θ ← InferSchedule(F)          // tiles, reduction axes, cores,
  aie_tile_size/aie_cores
// Phase 2: ReconstructDAG
V ← realized Func stages of F
E ← ∅
foreach producer p ∈ V read by consumer c ∈ V do
    add edge e : p → c to E
    halo(e) ← AccessWindow(c, p)
foreach e ∈ E do
    class(e) ∈ {p2p, broadcast, split, join, stencil}
return (Θ, V, E)
```

the loop nest for tile sizes, reduction axes, parallelization extent, core count, and any `aie_*` overrides. ReconstructDAG then takes the realized `Func` stages as nodes V and adds an edge $e : p \to c$ whenever stage c reads producer p, with $halo(e)$ the min/max offsets of c's accesses to p. A last pass labels each edge with an ObjectFIFO pattern (point-to-point, broadcast, split, join, stencil). Together with Θ's reduction axes, that labeling selects the workload's multi-core distribution—elementwise, multi-stage pipeline, reduction, or tiled contraction—detailed in §3.3. The emitter (§3.3) consumes (V, E) in memory; the autoscheduler (§4) consumes Θ by writing schedule directives that the same intercept reads back. Figure 3 traces the DAG half end to end for the blur→sharpen pipeline of Listing 2.

Why the lowered IR.. AIEHalide intercepts after bounds inference, which has already materialized the per-tile regions, halos, and L1/L2 working-set sizes that the hard capacity constraints of §4 need. Buffers are kept multi-dimensional through lowering, so the multi-dimensional indices that DMA descriptors and layout selection require remain intact rather than being collapsed to flattened one-dimensional accesses. Intercepting earlier forgoes bounds inference; reading the default flattened form would force the graph to be rebuilt from one-dimensional loops.

(nodes, edges, and halos via `analyze_access_patterns`) then `detect_dataflow_patterns` (edge classification).

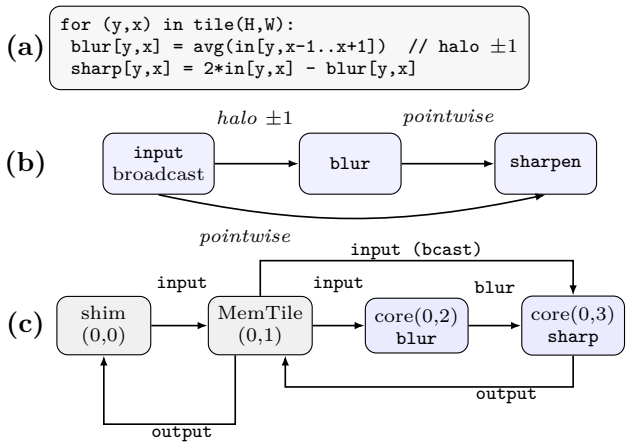


Figure 3: Reconstructing and lowering the blur→sharpen pipeline of Listing 2: (a) bounds-inferred loop nest with the ±1 halo; (b) recovered DAG (`input` broadcast to both stages, with halo and pointwise access patterns); (c) emitted AIE dataflow on one column—one `aie.core` per stage and five `aie.objectfifos`: the input is read from L3 once and broadcast at the MemTile to both consumers via a single multi-consumer ObjectFIFO, over linked L3↔L2↔L1 staging.

3.3 Dataflow Emission

Mapping DAG to dataflow MLIR.. A unified graph builder emits dataflow MLIR through a direct correspondence: each materialized `Func` becomes an `aie.tile` + `aie.core` pair; each inter-core edge becomes an `aie.objectfifo` streaming channel whose per-slot size follows the tile dimensions and depth follows the schedule's δ. Host edges are staged as linked L3↔L2 and L2↔L1 ObjectFIFOs, with the L3 transfer programmed by an `aiex.npu.dma_memcpy_nd` op using multi-dimensional strided access.

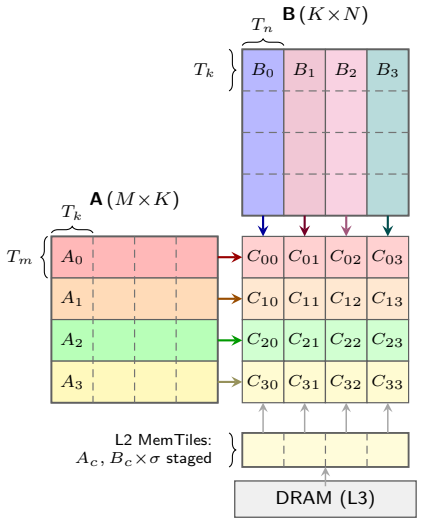


Figure 4: Output-stationary GEMM mapping (shown for 4×4). Each cell is a compute tile (L1) that accumulates the matrix-C tile C_{rc} as row band A_r broadcasts horizontally and column band B_c broadcasts vertically. L2 MemTiles (one per column) stage σ K-slices ($A_c, B_c \times \sigma$) for sustained DRAM throughput; dashed lines inside A and B mark T_k slice boundaries.

Multi-core distribution. The DAG-to-core mapping is derived from the schedule—`parallel` loops, reduction variables, and the index expressions on each producer–consumer edge; the four cases below cover every operator class our test suite exercises.

- **(1) Elementwise** (no reduction): a parallelized output dimension partitions output tiles across cores round-robin; each core reads its own input slice with no inter-core traffic.
- **(2) Multi-stage pipelines** (e.g., the unsharp-mask chain blur → difference → sharpen): each `Func` node is assigned to one or more cores and producer→consumer DAG edges become `aie.objectfifo` streaming channels; `aie_fuse_with()` optionally places a producer–consumer pair on the same AIE tile, eliminating inter-tile DMA.
- **(3) Reductions with parallel output** (e.g., softmax, layernorm, row-wise normalization): the parallelized output axis partitions output slices across cores; each core computes its slice's reduction independently, with no inter-core data movement during the reduction, and per-core results land in their respective output positions through DMA writeback at the correctly assigned offsets—no explicit combine stage is required.
- **(4) Tiled contractions** (GEMM-class, parallel output *and* reduction): the parallelized output axes induce an *output-stationary* 2D distribution (Figure 4). The core at grid position (r, c) holds output tile C_{rc} ($T_m \times T_n$) resident in L1 throughout the K-reduction; A tiles ($T_m \times T_k$) are broadcast across the cores in grid row r and B tiles ($T_k \times T_n$) across the cores in grid column c, so each core accumulates $C_{rc} = \sum_k A_{rk} \cdot B_{kc}$ as A and B tiles stream in.

Buffer depths and L2 staging. Each streaming channel gets a default buffer depth δ from its access pattern—stencils use the stencil window plus one slot, GEMM defaults to double-buffered inputs and a single-buffered output ($\delta_A = \delta_B = 2, \delta_C = 1$); `aie_dataflow()` or the autoscheduler emits explicit overrides. The L2 staging factor σ acts one level up: each shim-DMA descriptor batches $K_{mt} = \sigma \cdot T_k$ K-slices into a single DRAM→L2 transfer, amortizing per-transfer setup overhead at the cost of L2 footprint, bounded by the 512 KB MemTile capacity per column. Both δ and σ trade L1/L2 occupancy against larger tile dimensions; the autoscheduler explores this tradeoff as part of the search space (§4).

Layout transformation via 4D DMA descriptors. A core contribution of the compiler is realizing data-layout reformats as *DMA descriptor configurations rather than compute kernels*. Each MemTile DMA descriptor exposes up to four `(size, stride)` pairs through the `dimensionsToStream` attribute (`AIEDeviceModel::DMA_NDIMS=4`), describing how *any* operand is walked from L2 and deposited into L1.

Reordering that walk is what selects an operand's layout: a row- vs. column-major choice is a permutation of the `(size, stride)` pairs, not a transpose kernel. In the GEMM running example this is what flips B between contiguous-j and contiguous-k, but the same descriptor reorder applies to any operand of any operator. For contraction kernels a four-dimensional walk additionally produces the *block-interleaved layout* that `aievec.matmul`'s register tiling expects; elementwise and reduction kernels reuse the same machinery with a simpler, lower-dimensional walk. Every such reformat is derived from the Halide schedule (and, for contractions, the selected `mmul` shape) and incurs *no compute-side cost*—the rearrangement happens *during* the L2→L1 stream, ahead of the broadcast to the compute rows. Prior NPU and spatial-accelerator work typically dedicates compute passes (or pre-processing on the host) to such layout reformats; here they reduce to descriptor configuration alongside everything else the dataflow path emits. Most of these schedule choices stay within the dataflow path; the kernel and host paths see only a small subset, a locality we quantify in §3.6 (Table 3).

3.4 Compute Kernel Compilation

Compute kernels reach AIE tiles through two paths (Figure 2): the default *auto-generated path* lowers Halide expressions through linalg, transform, and AIEVec to LLVM IR; an *external-kernel path* attaches a hand-tuned C++ kernel via `aie_kernel(path)` where hand tuning still wins.

Auto-generated kernels. When no external kernel is specified, the compiler derives an MLIR kernel directly from Halide IR. For contractions, `AIEAutoTransform` emits a register-tiled `vector.contract` kernel that `aie-opt` lowers to `aievec.matmul`; for elementwise and reduction stages it emits `affine`/`memref` loops that lower through the standard AIEVec pipeline. Peano [4] compiles the AIEVec output to per-core ELF. Two cases still rely on external kernels for peak throughput: GEMM contractions on harder tile choices, where the register-tiling heuristic (used to minimise vector loads/stores) leaves a gap to hand-tuned `aie::mmul`; and reductions with transcendentals (`softmax`, `gelu`), where AIEVec lowering of `exp`/reciprocal and `ups`/SRS (upshift/shift-round-saturate) on second-generation tiles is incomplete. These cases link standard `aie_kernels` from `mlir-aie` [3]; closing the gaps is active work. Dataflow and tiling are decoupled from kernel generation: attaching a hand-tuned kernel to the same autoscheduled tiling recovers the headline peaks, which localizes the residual gap to vector-microkernel generation rather than to the tiling itself.

External kernels via `aie_kernel`. This directive attaches a pre-compiled C++ kernel to a dataflow node via the emitted MLIR's `link_with` attribute; any kernel matching the tile-sized `aie.objectfifo` interface plugs in without changing the surrounding dataflow graph. The headline GEMM numbers in §5 use C++ microkernels built from AIE-API `aie::mmul` intrinsics—the same library that backs IRON [9] and StB [18].

Data types, layout, and stability. Buffer declarations select the matching catalog `mmul` shape for int8, int16, bf16, fp16, and mixed precision (e.g., int8×int4; Table 4). Both paths consume operands in whichever layout the dataflow DMA descriptors established (§3.3); no in-kernel rearrangement is needed. The kernel path itself responds only to tile sizes (T_m, T_n, T_k) and the `mmul` shape $\langle r, s, t \rangle$—buffer depths, the L2 staging factor, layouts, core count, and tile-to-core mapping all stay in the dataflow path. An autoscheduler sweep therefore re-emits only the dataflow MLIR while reusing one compiled kernel binary.

3.5 Host Code Generation

The host path emits a C++ XRT application directly from the Halide pipeline's `Input<Buffer<T,N>` and `Output<Buffer<T,N>` declarations together with the runtime sequence emitted by the dataflow path; the user writes no host code. The application performs the standard XRT sequence (load `xclbin`, allocate DRAM buffers, push the runtime sequence to the shim controller, wait, read outputs back), and `aiecc.py` co-emits the runtime-sequence binding with the dataflow MLIR. IRON requires the developer to write this skeleton by hand and keep buffer sizes, types, and the instruction-binary path in sync with a separate Python design file (Listing 1); here the host binary tracks the dataflow automatically. Because it depends only on I/O dimensions, an autoscheduler sweep reuses one host binary across hundreds of schedule candidates.

3.6 Cross-Generation Portability

The device model abstraction. A centralized `AIEDeviceModel` encodes all per-generation hardware parameters (tile grid, L1/L2 capacities, DMA channel counts, vector width, microkernel shapes); the compiler reads it when emitting dataflow MLIR and selecting microkernel intrinsics, leaving the Halide source unchanged. The same generator targets XDNA (AIE2, 4×5) and XDNA 2 (AIE2P, 4×8); Table 2 lists the parameters that change between generations.

Table 2: Hardware parameters across AIE generations. The compiler updates `AIEDeviceModel` once per target.

Parameter	AIE2 (XDNA)	AIE2P (XDNA 2)
Compute tiles	4 col × 4 row	8 col × 4 row
`mmul` bf16	⟨4, 8, 4⟩	⟨4, 8, 8⟩
`mmul` i16	⟨4, 4, 4⟩	⟨4, 4, 8⟩
`mmul` i8	⟨4, 8, 8⟩	⟨8, 8, 8⟩
MACs/cyc (i8)	256	512
Accum. width	512-bit	2048-bit

Why this is cheap — schedule-decision locality. Most schedule decisions stay within the dataflow path (Table 3); the kernel path responds only to tile sizes and the `mmul` shape. The host path is independent of schedule, depending only on the problem's I/O dimensions. A generation change therefore touches the device model and dataflow path while leaving the kernel scaffolding and host application untouched. An autoscheduler sweep of hundreds of candidates likewise re-emits only the dataflow MLIR and reuses one compiled kernel binary.

Table 3: Schedule-decision locality across compilation paths. Most schedule choices localize to the dataflow path, leaving the kernel and host paths largely stable across schedules and across AIE generations. †When B column-major is fused into the kernel rather than realized as a 4D DMA descriptor, input layout reaches the kernel path.

Schedule choice	Dataflow	Kernel	Host
Tile sizes (T_m, T_n, T_k)	✓	✓	—
Buffer depths $(\delta_A, \delta_B, \delta_C)$	✓	—	—
Input data layout	✓	(†)	—
L2 staging factor σ	✓	—	—
`mmul` shape $\langle r, s, t \rangle$	—	✓	—
Core count C	✓	—	—
Tile-to-core mapping	✓	—	—

Cost comparison. Adding a new AIE generation requires ∼20 lines of device-model updates—$O(1)$ per generation—versus $O(K)$ for library-based approaches, where each of K operators must be rewritten for the new intrinsic shapes, register layouts, and data-packing rules (typically 50–200 lines per kernel). Even high-level APIs that use template metaprogramming to lift abstractions above raw intrinsics still require library authors to add new datatypes and tune for each new NPU (§5.4). Factoring this work into one place is what makes the *single-source two-generation* claim sustainable in practice.

4 NPU Autoscheduler

The autoscheduler casts NPU scheduling as constrained optimization. It searches a parameter space (tile sizes, core counts, buffer depths, layouts, etc.); rejects points that violate hard hardware limits (L1 capacity, vector alignment, DMA channel budgets, etc.); and scores the survivors with a small analytical model. Hard constraints eliminate enough of the space that what remains can be scored exhaustively.

Soft learned costs neither enforce compile-or-fail NPU limits nor avoid an unavailable training corpus. Hard-first pruning scores 59,694/120,960 GEMM candidates analytically in 150 ms without lowering; an NPU-specific learned model remains future work.

Each Halide pipeline is reduced to a loop-nest descriptor: dimensions tagged by role (parallel, reduction, batch) and buffers labeled with element width and dimension-access patterns. The search, hard constraints, cost model, and schedule application all run against this descriptor without branching on operation type. The same code path currently handles GEMM, Conv2D, element-wise maps, reductions (softmax, layernorm), attention, stencils, and fused pipelines.

Two examples run through the rest of the section: GEMM and softmax. GEMM is compute-bound with a large search space and reduction staging; softmax is DMA-bound, fused as a three-pass kernel (max, exp, normalize), with no staging. They exercise complementary corners of the framework.

4.1 Search Space

From the descriptor introduced above, the autoscheduler picks six parameters:

(1) **Tile sizes.** For each parallel and reduction dimension, the tile factor that determines how much data each core works on per invocation. GEMM: T_m, T_k, T_n. Softmax: a single T along the parallel dim.
(2) **Core count and grid shape.** How many compute tiles to use, arranged in R×S rows and columns.
(3) **Buffer depths.** Per channel: depth 1 (single) or 2 (ping-pong). Double buffering enables DMA/compute overlap; single buffering frees L1 for larger tiles.
(4) **L2 staging factor.** For reductions, how many slices along the reduction dimension are pre-staged into L2 per shim DMA transfer ($k_{mt} \in \{1, 2, 4, 8\}$). Set to T_k when no staging is used.
(5) **Microkernel shape.** $\langle r, s, t \rangle$ selected from the catalog (Table 4) by data type.

Algorithm 2: EnumerateAndPrune

Input: workload descriptor D (dims tagged parallel/reduction; buffers tagged by element width and access pattern); device model H
Output: feasibility-ranked candidate list (top-1 emitted; remainder retained as fallbacks)

$\mu \leftarrow \textsc{Microkernel}(D.dtype, H)$ // catalog lookup, not searched
$A \leftarrow \textsc{Axes}(D)$ // per-dim tiles, grid, depths, layouts; k_{mt} staging only if D has a reduction dim
$C \leftarrow \textsc{Product}(A)$
$F \leftarrow \{\, c \in C \mid \text{C1–C6}(c, D, H, \mu) \,\}$
return $\textsc{Sort}(F,\ \textsc{Cost}(\cdot, D, H, \mu))$

(6) **Data layout and placement.** Per operand: row-major vs. column-major. Per tile assignment: contiguous vs. interleaved across cores.

Table 4 lists the hardware multiply-accumulate shapes supported on each device. Restricting tile sizes to catalog-compatible shapes prunes 3–5× more candidates than unconstrained enumeration.

Table 4: Microkernel shapes and throughput.

A×B	AIE2 $\langle r,s,t\rangle$	AIE2P $\langle r,s,t\rangle$	MACs/cyc AIE2	MACs/cyc AIE2P
i16×i16	4,4,4	4,4,8	64	128
i8×i8	4,8,8	8,8,8	256	512
bf16×bf16	4,8,4	4,8,8	128	256

Most combinations are infeasible. Algorithm 2 shows the complete search skeleton.

The axis set $\textsc{Axes}(D)$ is derived entirely from the descriptor—reduction staging (k_{mt}) and column-major repacking appear only when D carries a reduction dimension, and the microkernel shape follows from the element type—so one skeleton serves every operator without branching on operation identity. Unlike Halide beam search (soft CPU cost) or TVM/Ansor (measured trials), hard pruning precedes ranking; search never lowers candidates.

4.2 Search Space Pruning

C1. **L1 capacity.** All statically placed buffers must fit in 64 KB.
GEMM: with inputs A, B and accumulator C,

$$\delta_A \cdot T_m T_k d \;+\; \delta_B \cdot T_k T_n d \;+\; \delta_C \cdot T_m T_n a \;\le\; B_{\text{L1}} \tag{1}$$

where d is the input element width, a the accumulator width, δ the buffer depth (1 or 2), and B_{L1} is 64 KB minus stack and vector-alignment reserves.
Softmax: three buffers — input ($4T$ bytes, double-buffered), output ($2T$), and f32 exp scratch ($4T$) — plus a 1 KB stack reservation give $10\,T + 1{,}024 \le 65{,}536$, so $T \le 6{,}451$, and the largest power-of-two tile that fits is $T = 4{,}096$.

C2. **Vector alignment.** Tile dimensions must be multiples of the microkernel or SIMD alignment for the operation's compute shape.
GEMM: for i8 with an 8×8×8 mmul, T_m, T_k, T_n must each be a multiple of 8.
Softmax: the bf16 tile T must be a multiple of 8; candidates below the bf16 vector width (32 on XDNA, 64 on XDNA 2) are also dropped to avoid under-using the SIMD lanes.

C3. **Divisibility.** The problem extent along each tiled dimension must be a multiple of its tile size: $T_m \mid M$, $T_k \mid K$, $T_n \mid N$ for GEMM; $T \mid N$ for softmax.

C4. **Enough parallelism.** The product of parallel-dimension tile counts must be at least the core count RS: $(M/T_m)(N/T_n) \ge RS$ for GEMM; $N/T \ge RS$ for softmax.

C5. **DMA channels.** Each column has one MemTile with six MM2S (memory-mapped-to-stream) and six S2MM (stream-to-memory-mapped) channels. For r cores in a column with n_{in} inputs and one output (one input distributed across cores, the rest broadcast), the MemTile uses $r + n_{\text{in}}$ channels in each direction — both must stay within six, so $r \le 6 - n_{\text{in}}$. Apply per column; treating it as a global cap leaves all but one column idle.
GEMM: with $n_{\text{in}} = 2$ (matrices A, B), $r \le 4$ — exactly the four physical rows.
Softmax: with $n_{\text{in}} = 1$, $r \le 5$; the four physical rows bind first.

C6. **L2 capacity.** Staged buffers in each MemTile must fit within 75% of its 512 KB L2 (≈ 384 KB), leaving room for control structures.
GEMM: the A slice is split across r rows ($r \cdot T_m \cdot k_{mt}$ elements total) and the B slice is broadcast ($T_n \cdot k_{mt}$ elements); double-buffered:

$$2d \cdot k_{mt} \cdot (r \cdot T_m + T_n) \le 384{,}000 \text{ bytes.} \tag{2}$$

Non-staged schedules (k_{mt}=T_k) satisfy this trivially.
Softmax: no reduction means no L2 staging; trivially satisfied.

Across C1–C6, 51% of GEMM's 120,960 candidates are rejected (§4.4); all 168 softmax candidates pass through (§4.5).[2] Pruning power scales with op complexity.

4.3 Cost Model

Each surviving candidate is scored as the maximum of four time components, plus an overhead term:

$$T = \max\left(T_{\text{compute}},\ T_{\text{L2}\to\text{L1}},\ T_{\text{DRAM}},\ T_{\text{shim}}\right) + T_{\text{overhead}} \tag{3}$$

Compute time. For GEMM, this is the number of microkernel calls per core multiplied by their cycle cost. For softmax, it covers three passes over the tile (max-reduce, exp-and-sum, normalize), with the exp lookup costing roughly 10× a simple vector op.

DMA time (L2→L1). Per-core bytes divided by calibrated sustained bandwidth (48 GB/s aggregate on XDNA 2, 32 GB/s on XDNA), including strided/irregular transfers. This is the bottleneck for softmax: each 512-element bf16 tile moves 1 KB in and 1 KB out.

DRAM time. Total bytes moved from system memory, divided by DRAM bandwidth (~40 GB/s on XDNA 2). For GEMM, reuse across the reduction dimension reduces this; for softmax, every element is read and written exactly once, so DRAM time scales linearly with N/C.

SHIM time. Per-column DRAM traffic through the SHIM DMA channels. Without contention this matches T_{DRAM}; a 3% penalty per column beyond two captures shared memory-controller pressure (~18% at full XDNA 2 deployment with eight columns). C5 hard-caps six channels per direction per MemTile; dynamic NoC arbitration between active routes is not modeled.

[2]The search additionally drops candidates whose per-core kernel exceeds tile instruction memory (an IMEM check alongside C1–C4). On GEMM, C5's six-channel cap coincides with the four physical rows and is enforced at code generation rather than pruning additional candidates here.

Overhead. Overhead has two parts: a per-tile-iteration cost (~3 μs per tile on XDNA 2, calibrated against hardware) and DMA descriptor setup. The total therefore scales with the number of tile iterations.

The model picks out the *bottleneck* for each candidate. GEMM is compute-bound at large sizes and DMA-bound at small sizes. Softmax is always DMA-bound: the per-element work (three passes, LUT-based exp) is cheaper than the memory transfer.

4.4 Walkthrough: GEMM

For a 4032×4320×4608 i8 GEMM on XDNA 2, the search starts with 120,960 raw candidates over tile sizes, core counts, per-channel buffer depths, and k_{mt} staging factors. C1–C6 reject 51%, leaving 59,694 to be scored. Sequential search completes in 150 ms on an AMD EPYC 9684X (Zen 4 + 3D V-Cache) host, compiled with g++-13 in -O3.

Table 5 shows the top-5 ranked candidates. All use 32 cores in 4×8 with T_m=144, T_n=192 and a single-buffered output (freed L1 lets T_k go larger). T_k ranges from 96 down to 48, k_{mt} takes 2× or 4×, and rank 5 ping-pongs the input buffer; measured latency spans 30%. Rank 1 wins because k_{mt}=2 keeps DRAM traffic burst-friendly and T_k=96 amortizes per-tile setup over more MACs; hardware confirms it as the empirical best at 25.46 TOPS.

Like TileLoom's [12] pre-profiling stage, our feasible list is analytically ranked; it preserves the measured top three (Spearman ρ=0.9, Kendall τ=0.8), picks the empirical best, and has 1.4× geometric-mean error. Profiling remains future work.

Table 5: GEMM top-5 of 59,694 feasible candidates (4032×4320×4608 i8, XDNA 2); all use 4×8 cores and $T_m \times T_n$ = 144×192.

Rank	T_k	k_{mt}	d_A	Latency	TOPS
1	**96**	**2×**	**1**	**6.30 ms**	**25.46**
2	80	2×	1	6.62 ms	24.24
3	72	2×	1	6.85 ms	23.42
4	48	4×	1	8.21 ms	19.54
5	48	4×	2	7.63 ms	21.05

Channel and core utilization. With two inputs (A, B) and one output (C), each MemTile uses all 6 MM2S channels (4 for L2→L1 split of A, 1 for L2→L1 broadcast of B, 1 for L2→L3 output of C) and all 6 S2MM channels (1 each for L3→L2 input of A and B, 4 for L1→L2 join of C). GEMM saturates the channel budget, so C5 and the 4-row physical limit both cap r at four. All 8 MemTiles, all 32 compute tiles, and 96 of 96 DMA channels are in use.

4.5 Walkthrough: Softmax

For a 262,144-element bf16 softmax on XDNA 2, the search generates 168 candidates, all of which clear C1–C6 (small per-tile L1, single input/output). The cost model identifies DMA as the bottleneck for every candidate: three vectorized passes over a 512-element tile cost ~1.3 μs of compute, while the L2→L1 transfer takes ~4.2 μs.

Table 6 shows the best candidate at each core count. Latency drops with every doubling of cores because each added column brings its own MemTile and DMA-channel budget; the gap from 16 to 32 cores narrows because compute time per tile starts to bound latency. An exhaustive sweep over {cores, tile} ∈ {1, 4, 8, 16, 32}×{512, 1024, 2048, 4096} finds 99 μs as the empirical best (32 cores, tile 4096); the autoscheduler picks 103 μs (tile 512), a 4% gap that keeps the L1 footprint small.

Table 6: Softmax top candidate per core count (N=262,144 bf16, XDNA 2; tile 512, double-buffered).

Cores	Grid	Cols	Measured
1	1×1	1 of 8	770 μs
4	4×1	1 of 8	249 μs
8	4×2	2 of 8	158 μs
16	4×4	4 of 8	120 μs
32	**4×8**	**8 of 8**	**103 μs**

Channel and core utilization. With one input and one output, each active MemTile uses 5 of 6 MM2S channels (4 for the L2→L1 split across the 4 cores, 1 for the L2→L3 output) and 5 of 6 S2MM channels (1 for the L3→L2 input, 4 for the L1→L2 join). The 4-row physical limit binds before the channel limit, leaving one channel of headroom in each direction. The chosen schedule activates all 8 MemTiles, all 32 compute tiles, and 80 of 96 DMA channels.

The same schedule maps onto XDNA as a 4×4 grid using 16 cores and 64 of 64 channels. Measured latency is 338 μs, consistent with the ~3.4× DRAM-bandwidth gap between the two devices. Both runs verify against a float-32 reference (max relative error 3%, within bf16 three-pass precision).

Residual compile-time failures. If unmodeled NoC congestion or L1 fragmentation rejects a candidate, the ranked list supplies the next — a compile-time failure, never a wrong output. Backend feedback remains future work.

5 Evaluation

We evaluate AIEHalide along two axes: (1) operator-level throughput versus hand-tuned baselines, and (2) cross-generation portability between XDNA and XDNA 2.

5.1 Experimental Setup

Hardware. (1) **XDNA** (AIE2): a 4×5 array of 20 compute tiles at 1.25 GHz; the shim tile in one column is unavailable, so we use a 4×4 subset (16 cores). Peaks for this mapping: 10.24/2.56/5.12 TOPS (int8/int16/bf16). (2) **XDNA 2** (AIE2P): a 4×8 array of 32 compute tiles with 2048-bit accumulators; peaks 40.96/10.24/20.48 TOPS. Peaks computed as cores × MACs/cyc × f × 2. Reported throughputs are averages over 10 timed iterations after 5 warmups.

Host. The autoscheduler search runs sequentially on an AMD EPYC 9684X (Zen 4 + 3D V-Cache) host with 62 GiB RAM, compiled with g++ -O3.

Toolchain and methodology. mlir-aie [3] for MLIR compilation; Peano [4] or xchesscc [5] for tile code; XRT 2.17 for host runtime. The compiler supports auto-generated and external (`aie_kernel`) compute paths (§3.4). Auto-generated GEMM kernels currently trail hand-tuned ones by ~15–30% (active work); GEMM headline numbers therefore use AIE-API `aie::mmul` microkernels—the same library used by IRON [9] and StB [18]—to isolate the dataflow compiler's contribution.

5.2 GEMM Performance

All results are verified correct; the same Halide source compiles to both platforms by changing only the target string.

Table 7: GEMM throughput on XDNA (16 AIE2 tiles). "brow" / "bcol" = B row-major / column-major.

Size	Type	Tile (m×k×n)	Flow	TOPS	%Pk
Int8 (peak = 10.24 TOPS)					
4032	i8→i8	112×112×112	brow	5.42	53%
4096	i8→i16	64×64×64	brow	2.64	26%
4096	i8→i32	64×128×128†	brow	3.07	30%
Int16 (peak = 2.56 TOPS)					
4096	i16→i16	64×64×32	bcol	1.15	45%
4096	i16→i32	64×64×64†	brow	1.36	53%
Bf16 (peak = 5.12 TOPS)					
4096	bf16→bf16	64×64×64	brow	1.98	39%
4096	bf16→f32	64×64×64	brow	1.95	38%

Sizes are M=K=N unless shown as M×K×N. All rows compiled with Peano [4] (open-source). †Autoscheduler-selected tile.

Table 8: GEMM throughput on XDNA 2 (32 AIE2P tiles). Same Halide source as Table 7.

Size	Type	Tile (m×k×n)	Flow	TOPS	%Pk
Int8 (peak = 40.96 TOPS)					
4032×4320×4608	i8→i8	144×72×144†	brow, σ=3	25.46	62%
4096	i8→i8	128×128×128†	brow	22.13	54%
3840	i8→i16	96×64×96	brow	14.97	37%
4096	i8→i32	128×128×64†	brow	10.31	25%
Int16 (peak = 10.24 TOPS)					
4096	i16→i16	128×32×128†	brow	6.18	60%
2048	i16→i32	64×64×64†	brow	5.41	53%

Sizes are M=K=N unless shown as M×K×N. †Autoscheduler-selected tile. Brow preferred on XDNA 2: 8-column DMA strides exceed bcol limits at large K.

Tables 7 and 8 cover three data types per generation, with autoscheduler-selected tiles marked †. On XDNA 2, the headline result is 25.46 TOPS (62% of peak) for 4032×4320×4608 i8 GEMM with asymmetric 144×72×144 tiles and σ=3; i16 reaches 60% with 128×32×128. On XDNA, i8 reaches 53% at 112[3].

The autoscheduler discovers platform-dependent layouts: bcol works on XDNA, but brow is required on XDNA 2 because 8-column DMA strides exceed bcol limits at large K. It also produces asymmetric tiles (e.g., 144×72×144 on XDNA 2) that outperform square tiles by 19–68% on wider-accumulator types ($i8 \rightarrow i32$, $i16 \rightarrow i32$). The cost model attains 1.4× geometric-mean prediction error; the L2 staging factor σ contributes most — omitting it raises error to 2.3×.

5.3 Comparison with Prior Work

Table 9 compares AIEHalide against StB [18] (hand-tuned IRON), framing the trade-off explicitly: hand tuning achieves higher peak (66% on XDNA, 93% on XDNA 2 for i8) at the cost of expert effort and per-generation rewrites, while AIEHalide reaches 53% / 62% automatically and ports across generations with zero source changes. The 53/62% versus 66/93% delta comes from the microkernel path rather than the autoscheduled dataflow and tiling. MLIR-AIR [20] reports up to 78.7% (i16), 59.1% (i8), and 48.6% (bf16) on XDNA using auto-generated dataflow with hand-optimized MLIR-AIE kernels; on the same XDNA platform AIEHalide reaches 53% / 53% / 39% across these three dtypes, rising to 60% (i16) and 62% (i8) on XDNA 2. ARIES [24] reports 0.078 TOPS/core on Versal™ (i16, 20 cores; different SoC).

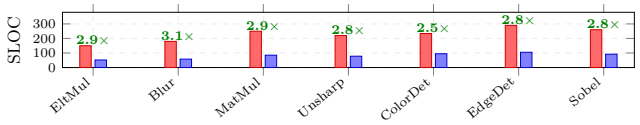


Figure 5: Source lines of code, hand-written IRON (red) vs. AIEHalide (blue), across seven representative benchmarks (shared infrastructure excluded). Green annotations show per-benchmark reduction; average 2.8× across the suite.

Table 9: GEMM comparison with StB [18] (hand-tuned IRON, k_{mt} staging, B col-major).

Work	Type	Size	TOPS	%Pk	Notes
XDNA (AIE2, 16 tiles, peak 10.24 TOPS int8)					
Ours	i8	4032	5.42	53	112×112×112, brow
StB	i8	best	6.76	66	—
XDNA 2 (AIE2P, 32 tiles, peak 40.96 TOPS int8)					
Ours	i8	4032×4320×4608†	25.46	62	brow, σ=3
StB	i8	best	38.05	93	—
Ours	i16	4096	6.18	60	128×32×128†, brow

Sizes are M=K=N unless shown as M×K×N. †Autoscheduler-selected tile.

5.4 Beyond GEMM: Vision, ML, and Productivity

Table 10 validates cross-generation portability on a subset of our 43-design suite (19 vision, 24 ML); all listed designs compile and pass functional verification on both platforms with **zero source changes**. Profiled XDNA 2 activations and elementwise kernels are sub-millisecond; convolutions span 217 μs–1.5 ms. Three-pass bf16 softmax uses all 16/32 cores on XDNA/XDNA 2 by replicating a shim→MemTile→ 4-core split per column, enforcing C5's six-channel limit locally rather than globally.

Table 10: Cross-platform coverage: one Halide source on XDNA (4×4) and XDNA 2 (4×8).

Design	Workload	XDNA	XDNA 2
blur two-stage	4096 elem u8	✓(731 μs)	✓(497 μs)
sobel	1920×1080 RGBA	✓(9,669 μs)	✓(5,286 μs)
edge detect	1920×1080 RGBA	✓(6,497 μs)	✓(4,785 μs)
color detect	1920×1080 RGBA	✓(3,005 μs)	✓(2,946 μs)
color six-stage	1920×1080 RGBA	✓(3,103 μs)	✓(3,075 μs)
threshold	1920×1080 RGBA	✓(2,558 μs)	✓(697 μs)
unsharp mask	1920×1080 RGBA	✓(2,928 μs)	✓(713 μs)
sigmoid bf16	65,536 elem	✓(710 μs)	✓(104 μs)
silu bf16	65,536 elem	✓(1,185 μs)	✓(103 μs)
gelu bf16[b]	65,536 elem	✓(1,181 μs)	✓(116 μs)
softmax bf16[a]	262,144 elem	✓(370 μs)	✓(103 μs)
relu bf16	65,536 elem	✓(417 μs)	✓(81 μs)
eltwise add bf16	65,536 elem	✓(515 μs)	✓(83 μs)
conv2d 1×1 i8	1024×64×64	✓(244 μs)	✓(217 μs)
conv3d 3×3×3 i8	216×1728×64	✓(3,919 μs)	✓(1,451 μs)

✓ = compiled and hardware-verified; parentheses give end-to-end latency (5 runs after 2 warmups). Conv shapes are N_{spatial}×C_{in}×C_{out}. [a]Three-pass bf16 softmax at maximal utilization: full 4×4 / 4×8 compute grid (16 cores XDNA, 32 cores XDNA 2), per-column split DMA, device-derived L1 tile (1024 XDNA, 512 XDNA 2). [b]GELU: Padé tanh approximation.

Compared to hand-written IRON, AIEHalide reduces source code by 2.8× on average across seven benchmarks (Figure 5; both counts exclude shared infrastructure). The same Halide source compiles to both XDNA and XDNA 2 with zero source changes.

6 Related Work

Hand-tuned GEMM and dataflow on AMD NPUs. IRON [9] exposes dataflow topology and DMA descriptors directly. Taka et al. [18] explore multi-level tiling, k_{mt} staging, and B column-major layout by hand; Wang et al. [19] add asymmetric tile buffering. AIEHalide derives the same infrastructure from Halide IR automatically.

DSLs and compilers for spatial NPU dataflow. MLIR-AIR [20] compiles affine loop nests to auto-generated AIE dataflow plus hand-optimized MLIR-AIE kernels, evaluated on XDNA for GEMM and LLaMA 2 attention. ARIES [24] needs explicit hardware mapping in the algorithm body and targets Versal™. CHARM [23] provides a higher-level model with explicit mapping; Triton-AIR [14] compiles Triton without algorithm–schedule separation. Interstellar [21] uses Halide schedules to design accelerators under an energy model, not to compile onto fixed, compile-or-fail silicon. TVM/Ansor [22] and prior Halide autoschedulers [1, 15] target cache-managed memory, where overflow degrades performance rather than failing compilation. Other Halide backends target FPGAs [10, 13], MPI [8], GPUs [6], a single AIE tile [7], or Versal stencils [17]; none targets spatial NPU dataflow. The FPGA path differs fundamentally: it *synthesizes* on-chip memory and interconnect, whereas AIEHalide maps onto a fixed array under hard capacity and channel budgets.

7 Conclusion

Halide's algorithm–schedule separation maps naturally to spatial NPU dataflow. AIEHalide adds constrained autoscheduling over tiles, cores, buffers, layouts, staging, and microkernels under hard L1/alignment/channel limits. Across 43 designs on both XDNA generations, XDNA 2 GEMM reaches 25.46 TOPS i8 (62% of peak) and 6.18 TOPS i16 (60%); auto-generated GEMM kernels trail hand-tuned C++ by ∼15–30%. Future work spans closing this microkernel gap, a hybrid cost model that profiles top-ranked feasible candidates, backend feedback for schedules that fail downstream, deeply fused attention and FFN pipelines, and retargeting the device model to other dataflow arrays.

References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 121:1–121:12. doi:10.1145/3306346.3322967

[2] AMD/Xilinx. 2023. *Versal ACAP AI Engine Architecture Manual (AM009).*

[3] AMD/Xilinx. 2024. MLIR-AIE: IRON API and MLIR-based AI Engine Toolchain. https://github.com/Xilinx/mlir-aie.

[4] AMD/Xilinx. 2024. Peano: Open-Source LLVM-based Compiler for AMD AI Engine. https://github.com/Xilinx/llvm-aie.

[5] AMD/Xilinx. 2025. Chess: Proprietary Compiler for AMD AI Engine Kernels. Proprietary backend bundled with Vitis™ and Ryzen™ AI toolchains. Selectable via `aiecc.py -aie-generate-chess`; also used in [18].

[6] Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, Steven Johnson, and Jonathan Ragan-Kelley. 2021. Efficient Automatic Scheduling of Imaging and Vision Pipelines for the GPU. *Proc. ACM on Programming Languages* 5, OOPSLA (2021), 1–26. doi:10.1145/3485530

[7] Prasanth Chatarasi, Hyunmin Siu, and Stephen Neuendorffer. 2020. Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine. *arXiv preprint arXiv:2006.01331* abs/2006.01331 (2020), 1–12.

[8] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. 2016. Distributed Halide. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM, New York, NY, USA, 5:1–5:12. doi:10.1145/2851141.2851157

[9] 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 *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM).* IEEE, Los Alamitos, CA, USA, 85–94.

[10] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, New York, NY, USA, 296–311. doi:10.1145/3192366.3192379

[11] 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 *Proc. International Symposium on Code Generation and Optimization (CGO).* IEEE, Los Alamitos, CA, USA, 2–14. doi:10.1109/CGO51591.2021.9370308

[12] Wei Li, Zhenyu Bai, Heru Wang, Pranav Dangi, Zhiqiang Zhang, Cheng Tan, Huiying Lan, Weng-Fai Wong, and Tulika Mitra. 2025. TileLoom: Automatic Dataflow Planning for Tile-Based Languages on Spatial Dataflow Accelerators. arXiv:2512.22168.

[13] Xinhao Li, Yun Liang, Shuo Li, Mingxing Yan, and Jason Cong. 2020. Hetero-Halide: From Image Processing DSL to Efficient FPGA Acceleration. In *Proc. International Symposium on Code Generation and Optimization (CGO).* ACM, New York, NY, USA, 51–63. doi:10.1145/3368826.3377916

[14] Eric Mahurin. 2024. Triton-AIR: Compiling Triton to AMD NPU. https://github.com/Xilinx/triton-air. Open-source effort.

[15] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 83:1–83:12. doi:10.1145/2897824.2925952

[16] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, New York, NY, USA, 519–530. doi:10.1145/2491956.2462176

[17] Gagandeep Singh, Alireza Khodamoradi, Kristof Denolf, Jack Lo, Juan Gómez-Luna, Joseph Melber, Andra Bisca, Henk Corporaal, and Onur Mutlu. 2023. SPARTA: Spatial Acceleration for Efficient and Scalable Horizontal Diffusion Weather Stencil Computation. In *Proc. ACM International Conference on Supercomputing (ICS).* ACM, New York, NY, USA, 463–476. doi:10.1145/3577193.3593719

[18] Endri Taka, Andre Roesti, Joseph Melber, Pranathi Vasireddy, Kristof Denolf, and Diana Marculescu. 2025. Striking the Balance: GEMM Performance Optimization Across Generations of Ryzen AI NPUs. arXiv:2512.13282.

[19] Chengyue Wang, Wesley Pang, Xinrui Wu, Gregory Jun, Luis Romero, Endri Taka, Diana Marculescu, Tony Nowatzki, Pranathi Vasireddy, Joseph Melber, Deming Chen, and Jason Cong. 2025. Can Asymmetric Tile Buffering Be Beneficial? arXiv:2511.16041.

[20] 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, Andre Rosti, Javier Setoain, Gagandeep Singh, Endri Taka, Pranathi Vasireddy, Zhewen Yu, Niansong Zhang, and Jinming Zhuang. 2025. From Loop Nests to Silicon: Mapping AI Workloads onto AMD NPUs with MLIR-AIR. arXiv:2510.14871.

[21] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* ACM, Lausanne, Switzerland, 369–383. doi:10.1145/3373376.3378514

[22] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Zhao, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Berkeley, CA, USA, 863–879.

[23] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. 2023. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).* ACM, New York, NY, USA, 153–164. doi:10.1145/3543622.3573210

[24] 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 *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).* ACM, New York, NY, USA, 1–11. doi:10.1145/3706628.3708870