



CONFERENCE

—NORTH AMERICA 2026—

Unlocking PyTorch for Dataflow Accelerators: An Open-Source Kernel Tile IR and Dataflow Scheduler

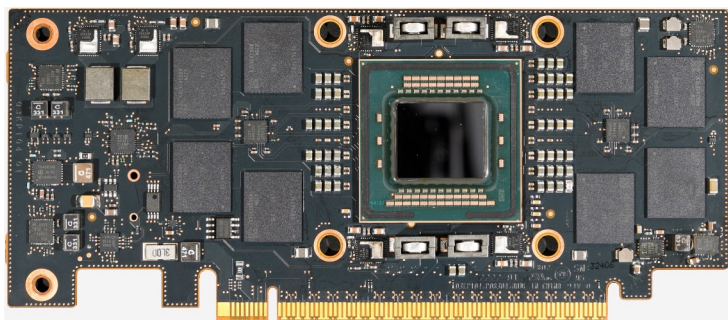
An open MLIR dataflow backend for the IBM Spyre accelerator

Prasanth Chatarasi, Viji Srinivasan, Swagath Venkataramani, and the IBM Spyre Team

IBM Research · PyTorch Conference 2026

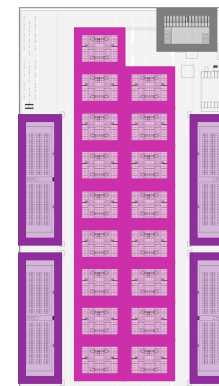
Spyre: Enterprise GenAI Inference Acceleration

Spyre I In production, 2025



- **75 W** PCIe Gen5 x16 adapter
- **32** RaPiD Gen4 AI cores
- **150+** fp8 TOPS, **300+** int8 TOPS
- **128 GB** LPDDR5 memory, **~200** GB/s bandwidth

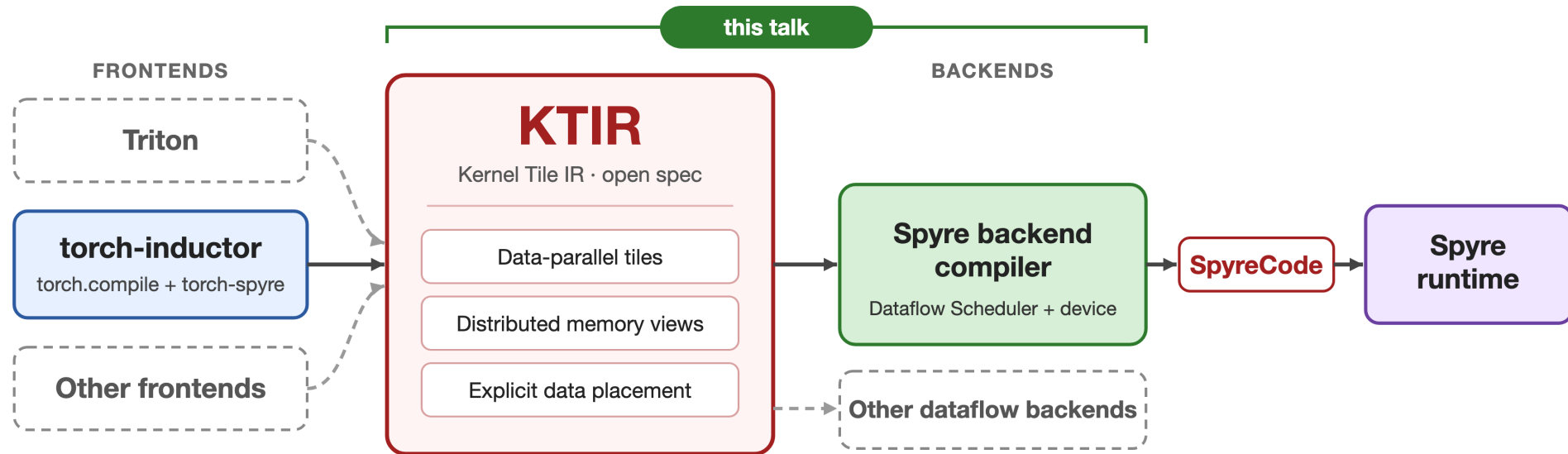
Spyre II Announced, 2026



- **16** RaPiD Gen5 AI cores, FP4/MXFP4
- **600+** fp8 TOPS, 4x current gen
- **96 GB** HBM3e, **~4** TB/s, 20x current gen
- **PCIe Gen6** x16, low-latency peer-to-peer

Designed for larger enterprise inference workloads

KTIR: A common interface for dataflow accelerators

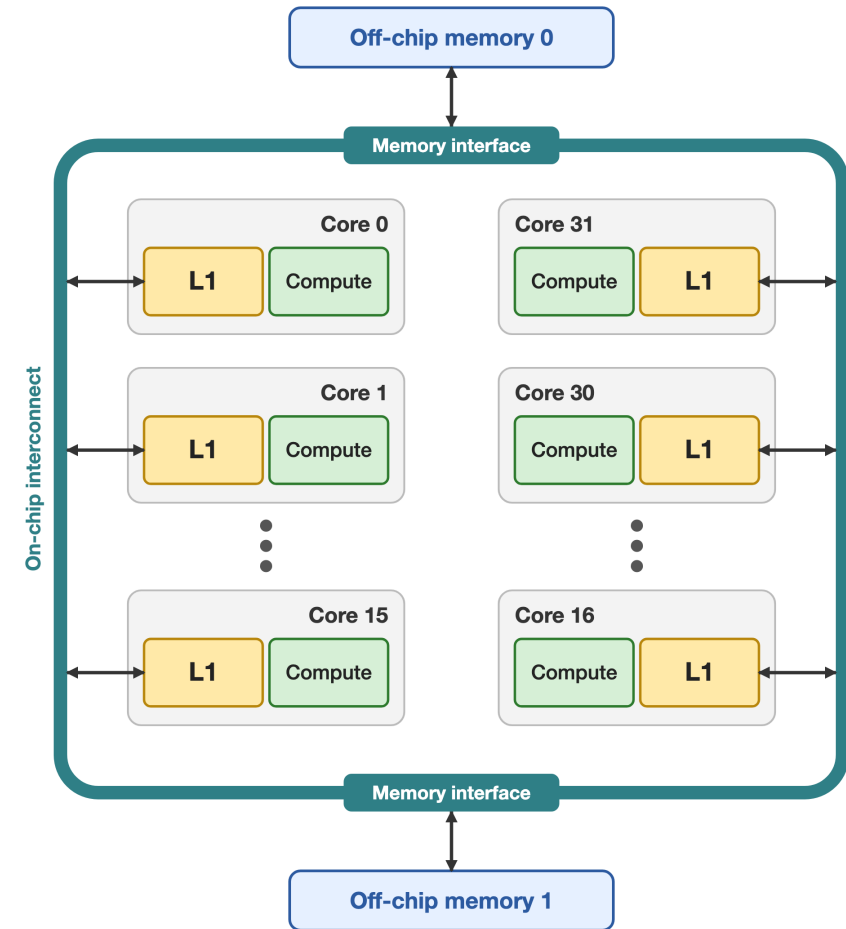


- Dataflow accelerators bring up unique challenges in the design of compiler frontend and backend interface
- Kernel Tile IR (KTIR): A tile-based IR designed to express “kernels” for multi-core dataflow accelerator architectures
- Many frontends, many backends. Torch-spyre with inductor emits KTIR today; the Spyre backend compiler generates SpyreCode for the runtime.

Dataflow Accelerators: Disaggregated Global/Local Memory

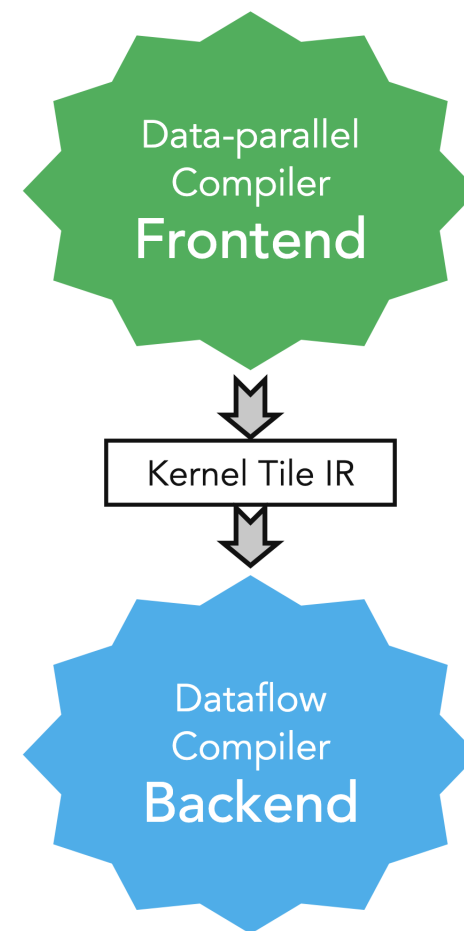
Memory-hierarchy trends

- **Distributed on-chip scratchpads**
 - Multiple levels of scratchpad memory
 - Private to a core, yet globally reachable
- **On-chip interconnect** moves data directly between local memories
- **Distributed off-chip memory** introduces intra-chip NUMA
- **The compiler** decides where tensors are resident and how compute accesses them
- **Common architecture trait** amongst many industry-leading AI accelerators

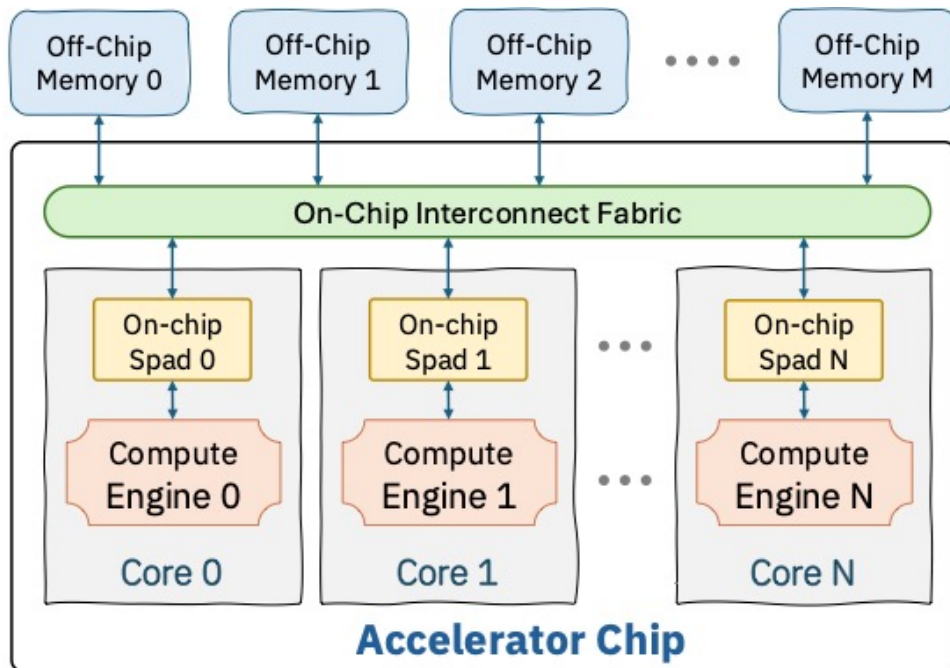


Kernel Tile IR (KTIR)

- **A tile-based interface** between the compiler frontend and backend for dataflow accelerators
- Expressive enough to capture data-parallel mappings from:
 - the PyTorch Inductor compiler frontend
 - custom Spyre kernels written in Triton or Helion
- **Uses compute, data and address tiles** to capture, independently:
 - work division across multiple cores
 - tensors accessed from distributed on-chip and off-chip memories
- **Represents multiple operators in one kernel**
 - operator composition with structured control flow
- **Built on open-source MLIR**
 - reuses existing MLIR dialects for tensor allocation, access, arithmetic and control flow



KTIR's data-parallel view of the accelerator

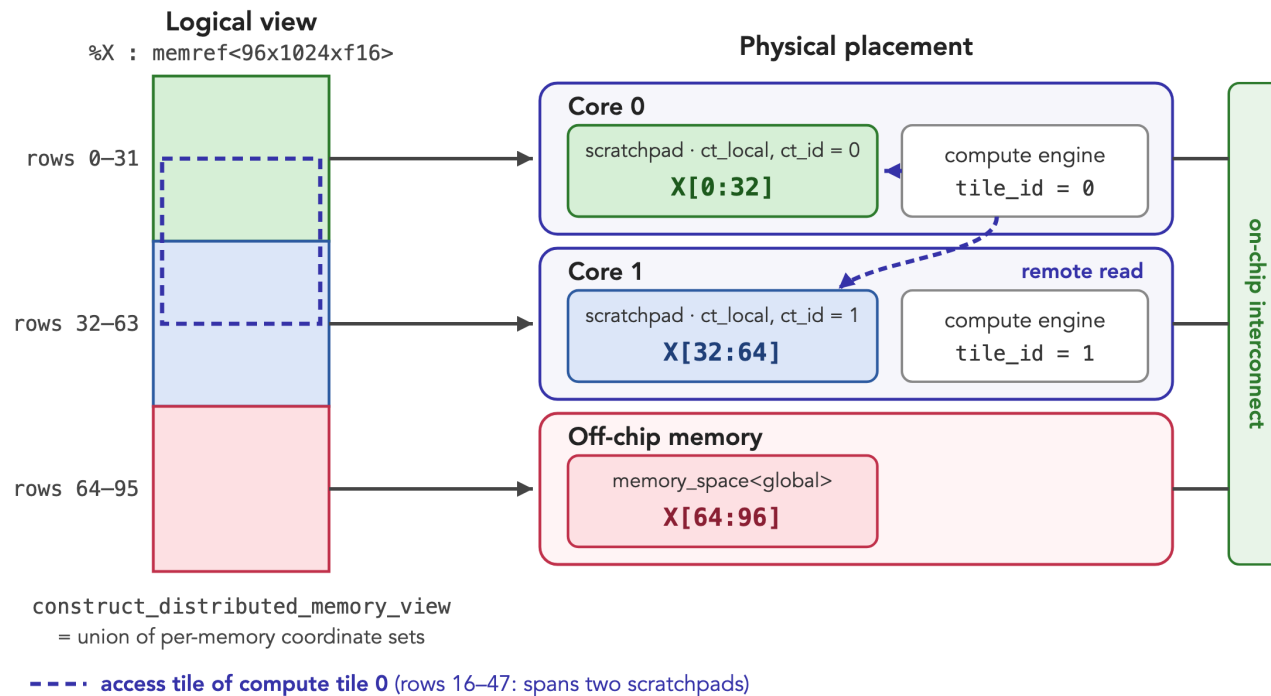


- Cores = **compute engine + its own scratchpad**
- **Every compute tile can see every memory**: its scratchpad, other cores' scratchpads, device memory
- KTIR programs describe computation mapping across cores and tensor residency across different memory
 - In KTIR, a core and tile are synonymous
 - 2 memory spaces: `ct_local` (scratchpad) and `global` (off-chip)
- Built from upstream MLIR: `arith`, `math`, `linalg`, `scf`, `tensor`, `memref`
- Two additional dialects: `ktdp` (memories + accesses) and `spyreop` (intrinsics)

Spec: torch-spyre RFC 0682 · github.com/torch-spyre/ktir-mlir-frontend

KTIR: Tensor Representation and Access

One logical tensor, distributed across two scratchpads and off-chip memory



- **One logical tensor**, composed from per-memory views
 - each piece lives in a core's scratchpad (`ct_local`) or off-chip (`global`)
- **Any compute tile can access any element**, local or remote, through access tiles
- **Access by logical coordinates**, constrained by integer sets
- **Layout decoupled from access**: the compiler can optimize data movement across memory spaces

Existing Tile IRs assume one unified off-chip memory reached through pointers; a tensor spread across on-chip scratchpads has no direct representation there.

KTIR operations

	Operation	What it does
Tensor layout	<code>ktdp.construct_memory_view</code>	Start address, sizes and strides of a tensor or sub-tensor in a given memory space (e.g., in one core's scratchpad)
	<code>ktdp.construct_distributed_memory_view</code>	Composes a full tensor that is split across multiple memory spaces (e.g., across several scratchpads)
Tensor addressing (access tile)	<code>ktdp.construct_access_tile</code>	Coordinates of a tensor that a tile needs to access
	<code>ktdp.construct_indirect_access_tile</code>	Coordinates a tile accesses indirectly, through a second index tensor
Tensor load / store (data tile)	<code>ktdp.load</code>	Reads a tile of a tensor using its layout (across memory spaces) and access tile
	<code>ktdp.store</code>	Writes a tile of a tensor using its layout (across memory spaces) and access tile
Inter-tile communication	<code>ktdp.inter_tile_produce / _reduce</code>	Exchanges partial results between compute tiles, e.g., an all-reduce
Compute	<code>linalg, arith, math + spyreop</code>	Arithmetic on data tiles, from upstream MLIR dialects plus Spyre intrinsics
Control flow	<code>scf.for, scf.if, scf.forall, ...</code>	How multiple tiles of work are sequenced and parallelized

A whole kernel: elementwise add over 32 tiles

```
// Addresses are arguments: PyTorch's allocator owns them
func.func @add(%A: index, %B: index, %C: index)
  attributes {grid = [32]} {
    %id   = ktdp.get_compute_tile_id : index
    %row0 = arith.muli %id, %c3 : index    // 3-row slab

    %A_view = ktdp.construct_memory_view %A,
      sizes: [96, 64], strides: [64, 1] {
        coordinate_set = #full,
        memory_space = #ktdp.memory_space<global>
      } : memref<96x64xf16>

    %A_t = ktdp.construct_access_tile %A_view[%row0, %c0] {
      access_tile_set = #tile, access_tile_order = #id
    } : memref<96x64xf16> -> !ktdp.access_tile<3x64xindex>

    %a = ktdp.load %A_t : ... -> tensor<3x64xf16>
    %b = ktdp.load %B_t : ... -> tensor<3x64xf16>
    %c = linalg.add ins(%a, %b ...) outs(%e ...)
    ktdp.store %c, %C_t : tensor<3x64xf16>, ...
  }
```

Four concerns, kept apart

Views: how memory is interpreted

Access tiles: which coordinates

load / store: the only data movement

linalg: the arithmetic

Work division is already settled

grid = [32] + the arithmetic on %id is the partitioning.

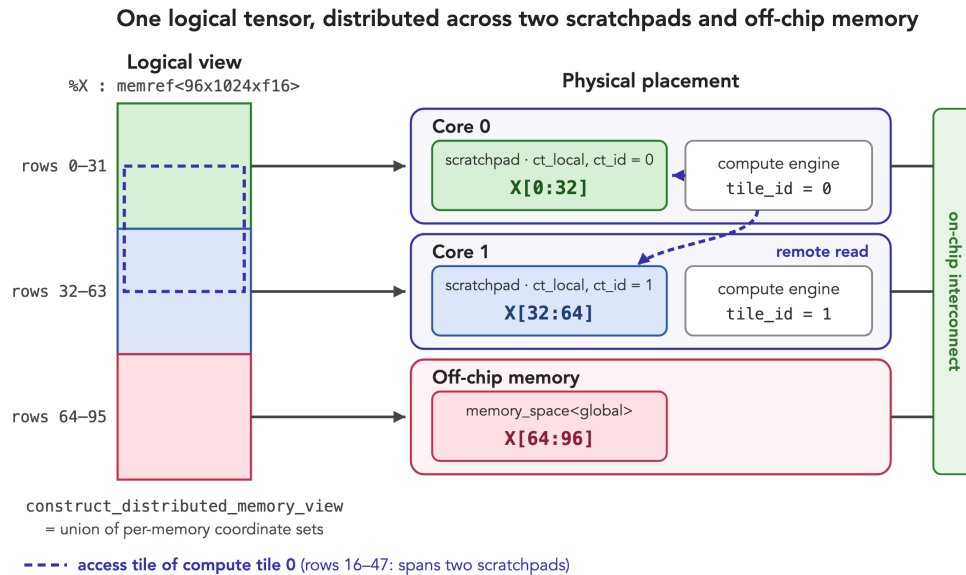
The backend honors it rather than second-guessing it.

Every Inductor decision has a home in KTIR

Inductor decides...	KTIR records it as...
Work division across cores	<code>grid attribute + arithmetic on get_compute_tile_id</code>
Stick-granularity layout	<code>memory-view strides + access-tile order</code>
Coarse tiling / working-set reduction	<code>scf.for</code> loops around <code>load</code> → <code>compute</code> → <code>store</code>
Scratchpad residency (address, core)	<code>a view in memory_space<ct_local, ct_id = k></code>
A tensor split across cores	<code>construct_distributed_memory_view</code>
Partial sums across cores	<code>inter_tile_produce / inter_tile_reduce</code>
Paged tensors and gathers	<code>construct_indirect_access_tile</code>
Activations, normalization steps	<code>spyreop</code> intrinsics inside <code>linalg.generic</code>
Tensor addresses and shapes	<code>kernel arguments / runtime_arg</code> – kept symbolic to <code>launch</code>

EXAMPLE

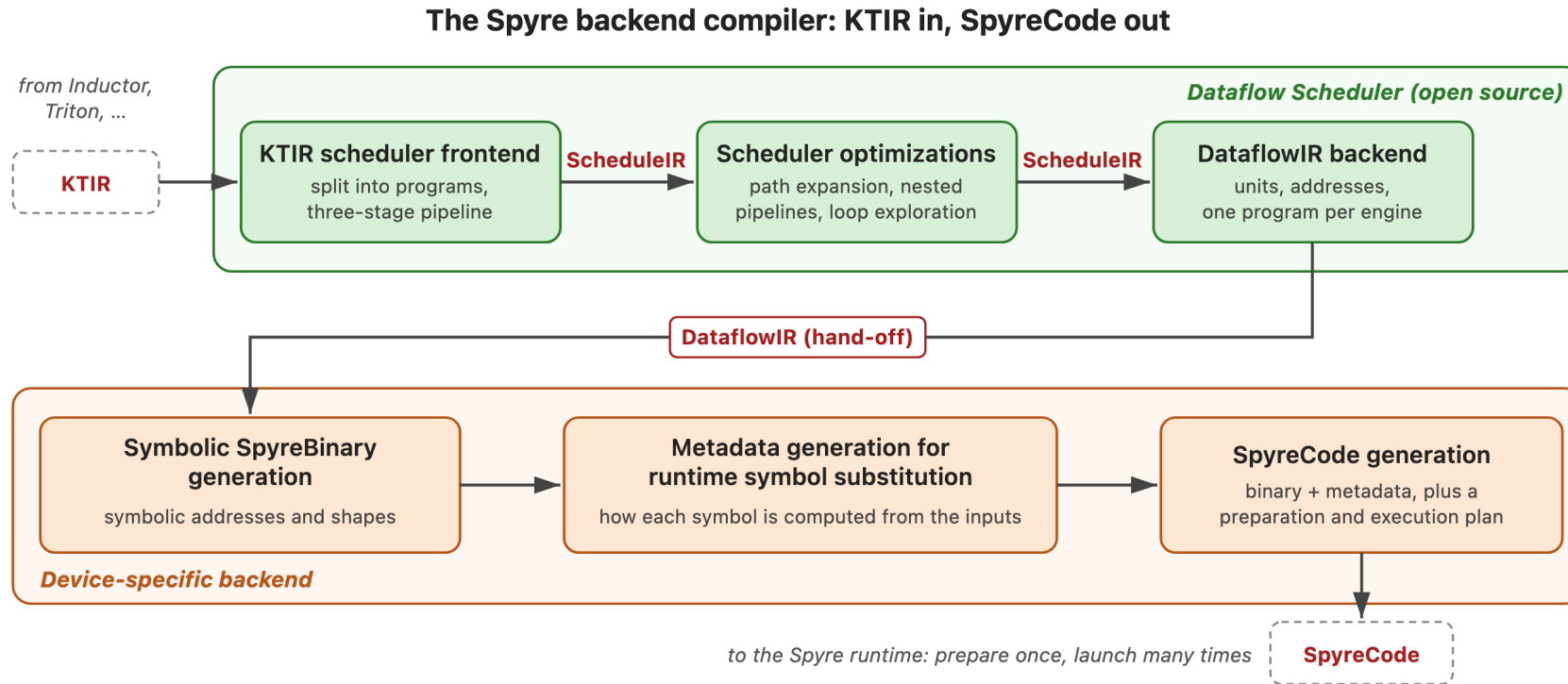
Tensors pinned in scratchpad, distributed across cores



- **KTIR:** on-chip placement is first-class
- **Intermediates stay on-chip** from one op to the next
- **Mismatched splits are just loads** across scratchpads
- **Placement is a binding contract** the compiled code depends on

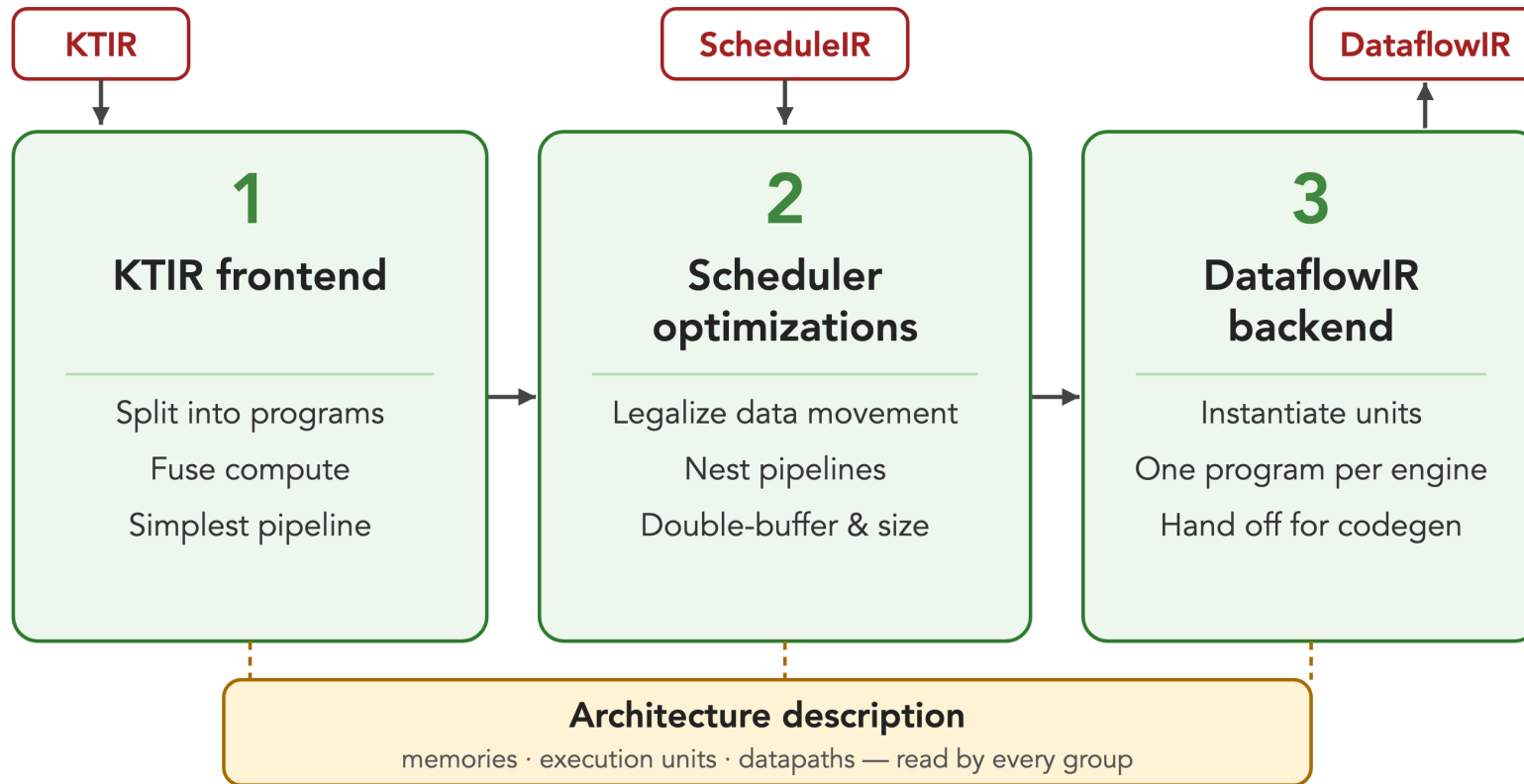
```
// X[96, 1024] f16: rows 0–31 in core 0's scratchpad, rows 32–63 in core 1's, rows 64–95 off-chip
%X0 = ktdp.construct_memory_view %off0, sizes: [32, 1024], strides: [1024, 1]
      { coordinate_set = #rows0, memory_space = #ktdp.memory_space<ct_local, ct_id = 0> } : memref<32x1024xf16>
%X1 = ... { coordinate_set = #rows1, memory_space = #ktdp.memory_space<ct_local, ct_id = 1> } : memref<32x1024xf16>
%X2 = ... { coordinate_set = #rows2, memory_space = #ktdp.memory_space<global> } : memref<32x1024xf16>
%X = ktdp.construct_distributed_memory_view (%X0, %X1, %X2 : ...) : memref<96x1024xf16>
// compute tile 0 reads rows 16–47: half from its own scratchpad, half from core 1's
%t = ktdp.construct_access_tile %X[%row, %c0] { ... } : memref<96x1024xf16> -> !ktdp.access_tile<32x1024xindex>
%x = ktdp.load %t : !ktdp.access_tile<32x1024xindex> -> tensor<32x1024xf16>
```

The backend compiler at a glance



- Open-source Dataflow Scheduler: KTIR → ScheduleIR → DataflowIR
- Device-specific stages turn DataflowIR into SpyreCode. DataflowIR is the only point where the two halves meet.

Dataflow Scheduler: scheduling means building a pipeline

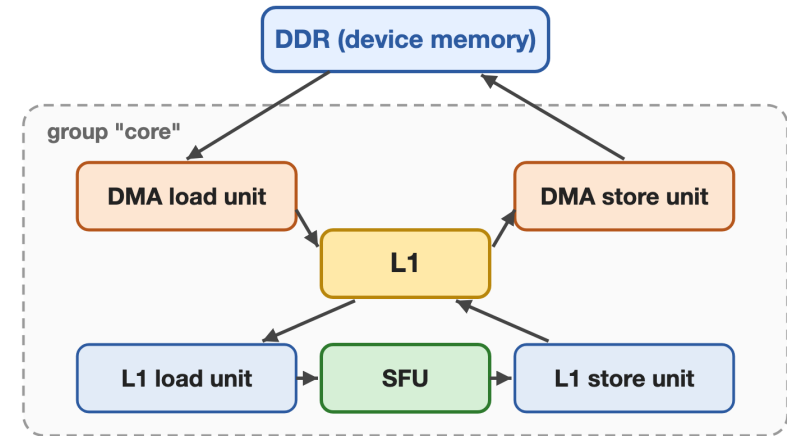


- **Three constructs:** loops, pipelines, parallel instances
- **Start simple, legalize, enlarge**
- **Three pass groups**, each with a fixed IR contract
- **Upstream MLIR** wherever possible
- **Architecture-driven**, not hard-coded

Hardware lives in an architecture description

```
#L1 = { kind = "L1", size = 1048576 } // 1 MB scratchpad
#SFU = { kind = "SFU", ktdf_arch.features = {
  ktdf_arch.feature.compute,
  ktdf_arch.feature.simd = {
    lanes = #ktdf_arch.map<f16 = 64, f32 = 32> } } }

ktdf_arch.device @sample_device {
  %ddr = memory #DDR
  group { kind = "core" } share(%ddr) {
    %l1 = memory #L1
    %mnilu = group {...} -> exec_unit // DMA load unit
    datapath %ddr to %mnilu : memory, exec_unit
    datapath %mnilu to %l1 : exec_unit, memory
    %sfu = group {...} -> exec_unit // compute unit
    %l1lu = exec_unit #L1LU // scratchpad load
    datapath %l1 to %l1lu : memory, exec_unit
    datapath %l1lu to %sfu : exec_unit, exec_unit
    ...
  }
  patterns ["pre_scheduling"] { ... } // device rewrites (PDL)
}
```



- **A graph:** memories and execution units as nodes, datapaths as edges
- **Datapaths** are the routing graph for path expansion
- **SIMD lanes, granularities, FIFO depths, sizes** shape every pipeline
- **PDL patterns** supply device lowerings
- **Retarget** = write a new description

KTIR Frontend: split into programs, build the simplest pipeline

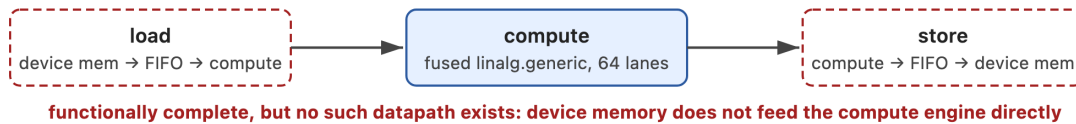
```
// ScheduleIR: an orchestrator calls one
// program per compute group
func.func @kernel() {
  scf.for %i = %c1 to %c5 step %c1 {
    func.call @local_schedule_1()
    scf.for %j = %c1 to %c3 step %c1 {
      func.call @local_schedule_0()
    }
    func.call @local_schedule_2()
  }
}
module @local_schedule_0 { ... }
module @local_schedule_1 { ... }
module @local_schedule_2 { ... }
```

- **Compute group** = one load → compute → store chain through memory
- **Each group becomes its own program** (nested module); the kernel becomes an orchestrator
- **KTIR loops stay outside** as program launches; loops inside a program belong to the scheduler
- **Each program then gets a three-stage pipeline:** load → compute → store over FIFOs, tiled to SIMD width
- Functionally complete — but not yet legal on the hardware

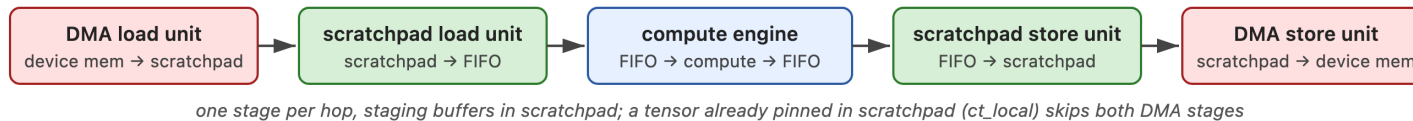
Scheduler Optimizations

Growing one pipeline: start simple, legalize, then enlarge

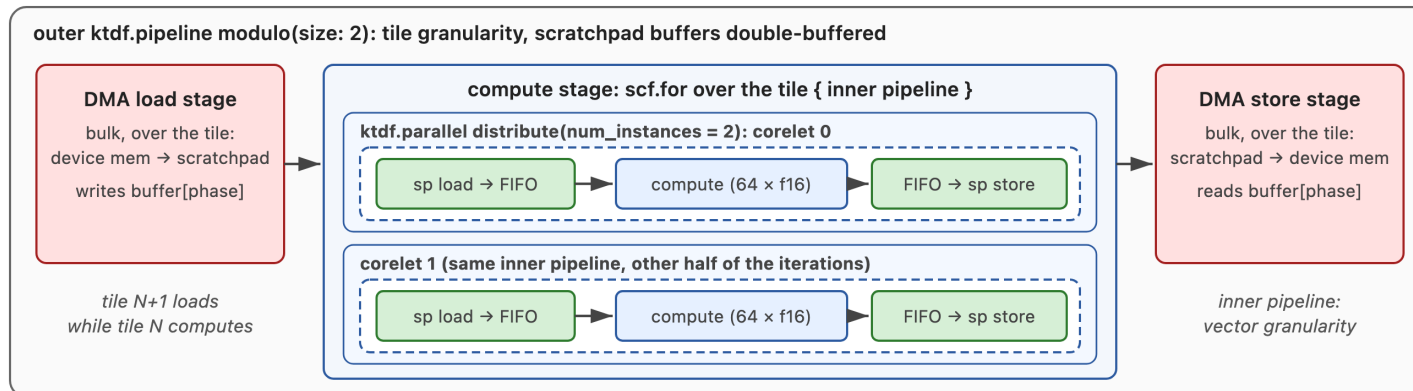
① the simplest pipeline: load → compute → store



② path expansion: route each hop over the architecture graph



③ tiling, nested pipelines, double buffering, corelet parallelism



- **Path expansion** routes each hop over the device graph
- **Pinned `ct_local` tensors skip DMA**
- **Nested pipelines:** bulk DMA outside, vector compute inside
- **Double buffering** overlaps load N+1 with compute N
- **Corelet split** adds parallelism inside a core

Every choice is explicit IR

Buffering depth → `modulo(N)`

Instance split → `distribute(N)`

Tile sizes → `reserve_size`

Each bounded by the architecture description

DataflowIR Backend

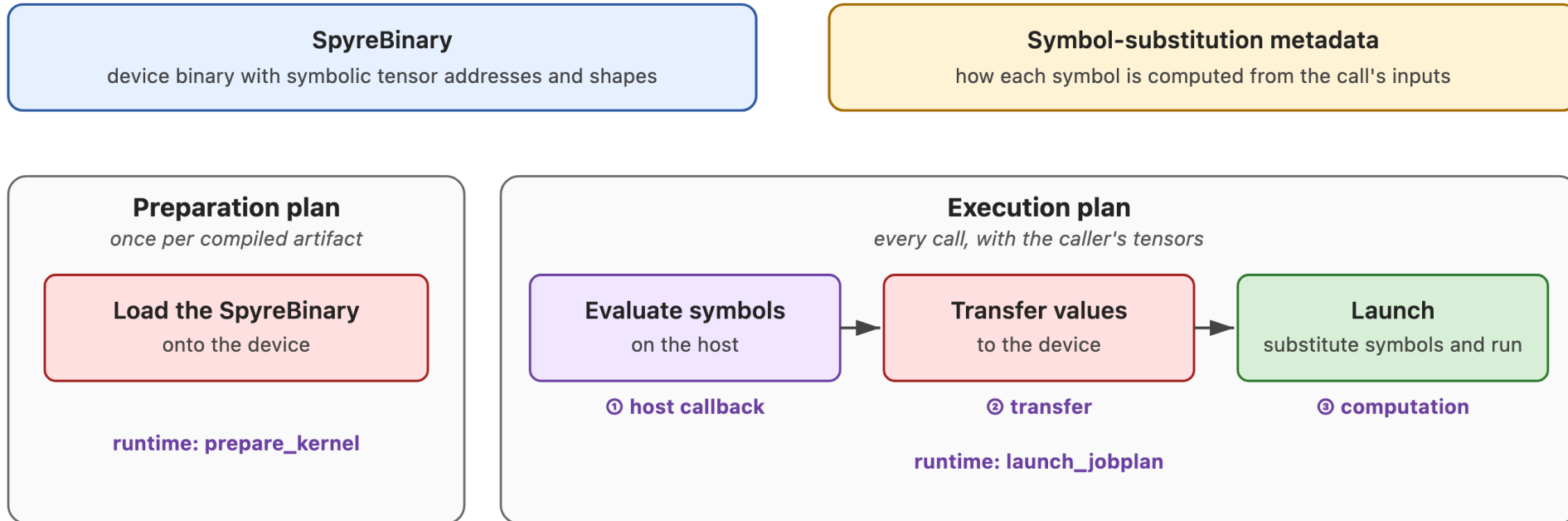
```
// One level up: what each symbol is
func.func @local_schedule_0(%base: index) {
  symbol.create_id %base {symbol_id = -1}
  %w0 = arith.divsi %base, %c64 : index
  symbol.create_id %w0 {symbol_id = -2} // grid elem 0
  %b1 = arith.addi %base, %c49152 : index
  %w1 = arith.divsi %b1, %c64 : index
  symbol.create_id %w1 {symbol_id = -3} // grid elem 1
  call @local_schedule_0_body() : () -> ()
}

// The DataflowIR only names symbols
%s0 = symbol.create_symbol {SymbolId = -2 : i64}
%s1 = symbol.create_symbol {SymbolId = -3 : i64}
%m = uniform.def_immutable_mapping(
  [%core0 -> %s0], [%core1 -> %s1]) : index
%a = uniform.query_map(map: %m, key: %unit)
dataflow.get_logical_memory_view %ddr, %a ...
```

- **Addresses assigned** against capacities in the description
- **Units instantiated** per core (and per corelet)
- One `dataflow.program_unit` per engine kind
- **Symbolic addresses and shapes survive:** each is an expression over the kernel's inputs
- **DataflowIR = the stable hand-off** to the device-specific backend

From DataflowIR to SpyreCode — and into the runtime

What SpyreCode carries



- **SpyreCode** = a SpyreBinary with symbolic addresses and shapes + symbol-substitution metadata + a two-phase job plan
- **Preparation plan** runs once (`prepare_kernel`); **execution plan** runs every call (`launch_jobplan`)

Where we are, and how to try it

Working today (V1)

Most LLM operations (softmax, layernorm, elementwise, reductions), from Inductor and Triton

Next

Matrix multiplication, a cost model, and end-to-end LLMs

```
git clone --recursive \
  https://github.com/torch-spyre/dataflow-scheduler.git
cd dataflow-scheduler
cmake -S . -B build -G Ninja \
  -DMLIR_DIR=$MLIR_INSTALL_PREFIX/lib/cmake/mlir
cmake --build build

# hello world: elementwise add on the sample device
./build/bin/dataflow-scheduler-opt -ktir-to-dfir \
  test/dataflow-scheduler-opt/ktir-to-dfir-sanity-check.mlir
```

Open source (Apache-2.0)

ktir-mlir-frontend: KTIR (ktdp + spyreop)
dataflow-scheduler: the scheduler and
sample architecture description
dataflow-scheduler-mlir-dialects:
ScheduleIR and DataflowIR dialects
RFCs: RFC 0682, the KTIR specification
All under github.com/torch-spyre

- Add `--mlir-print-ir-after-all` to watch every step
- Your own kernels and devices: `dataflow-scheduler -kEmitDFIR --device=<arch.mlir>`

Takeaways

KTIR

the interface

- **Kernels as data-parallel tiles:** work division, layout and tiling are explicit
- **On-chip placement is first-class:** tensors stay in scratchpad across ops
- **Open spec on upstream MLIR:** one target for Inductor, Triton and more

Dataflow Scheduler

the backend

- **KTIR to explicit pipelines:** path expansion, nesting, double buffering
- **Architecture-driven:** retarget by writing a new description
- **Open source:** hands symbolic DataflowIR to the device backend

Let's talk

Building a compiler for a dataflow or scratchpad-based accelerator?

We'd love to hear from you, and to see more architecture descriptions.

Read more on pytorch.org: [Building Spyre as a Native PyTorch Device](#) · [Extending Inductor for the IBM Spyre Accelerator](#)