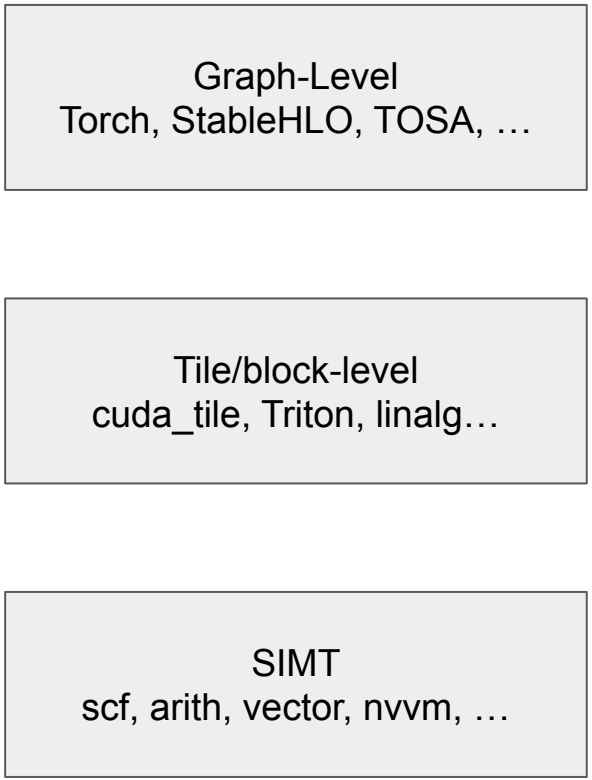
From Graphs to Warps: Semantic Interoperability Across MLIR Abstraction Levels

Nachiketa Gargi
NVIDIA

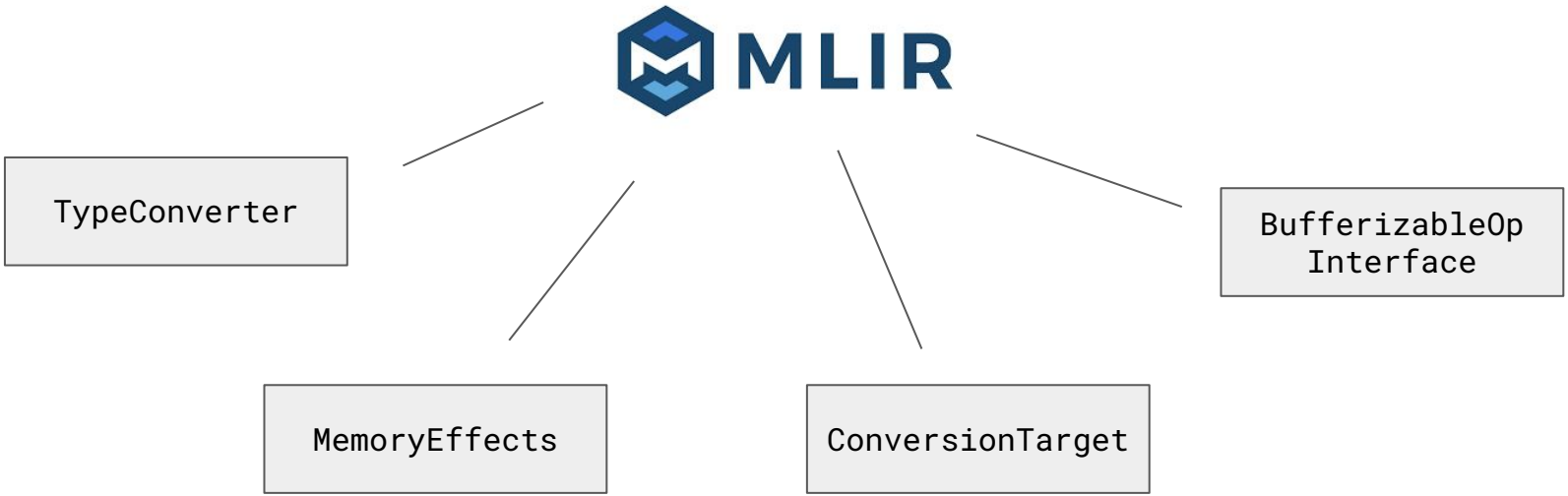
Overview

- High level compilers often require escape hatches to lower-level code
- Two case studies: graph- and tile-level
- Design space around contract

Semantic Interoperability?



The host compiler knows enough about foreign code at a different abstraction level to preserve correctness and still support meaningful optimization across the boundary.

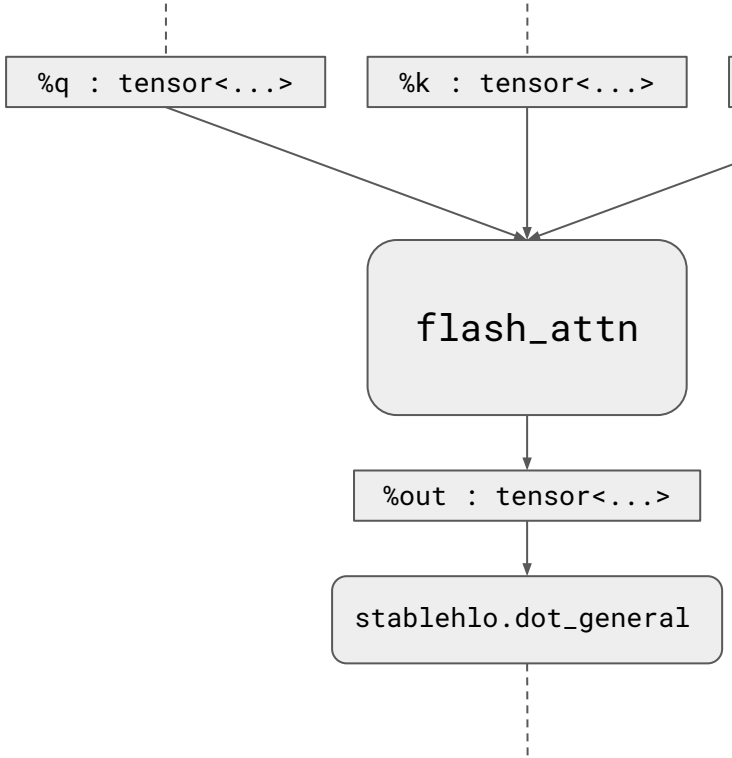

MLIR
TypeConverter
MemoryEffects
ConversionTarget
BufferizableOp
Interface

What is the minimal semantic summary that a host compiler needs about foreign code to make correct, performant decisions?

Graph Compiler Escape Hatch

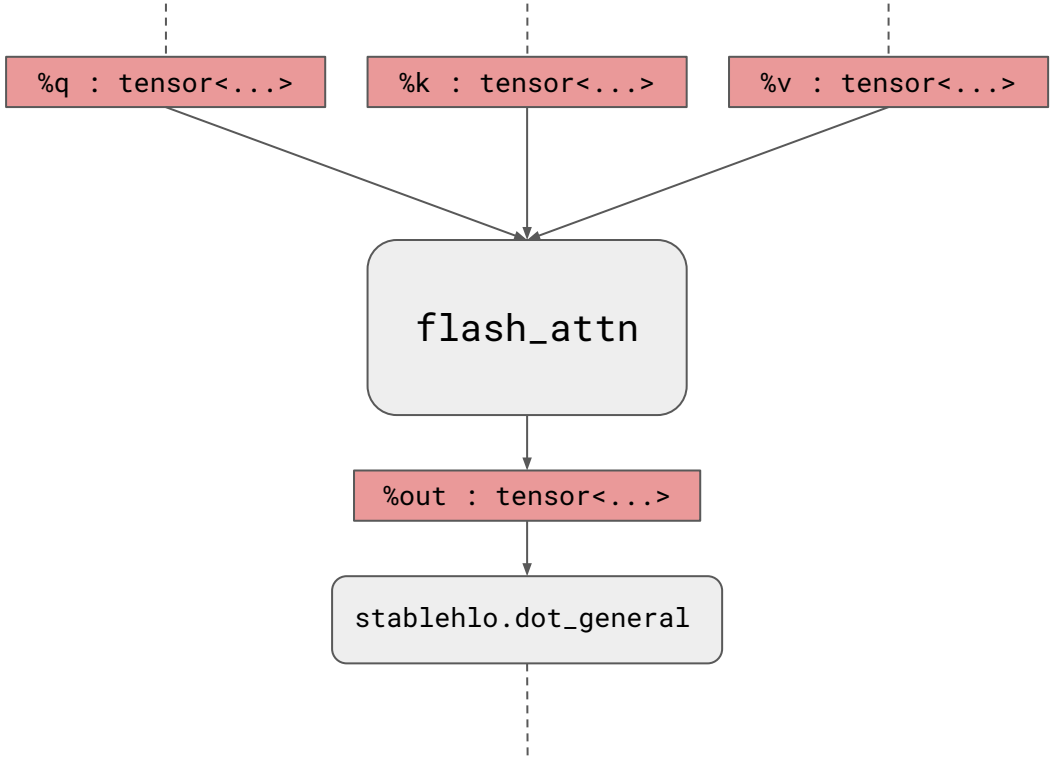
- A graph-level op that calls foreign code instead of being fully lowered by the host compiler
- Most modern ML graph compiler stacks support some sort of escape hatch
  - e.g. `stable_hlo.custom_call`, TensorRT Plugins, Modular MAX CustomOp
- Why?
  - Functionality
  - Performance

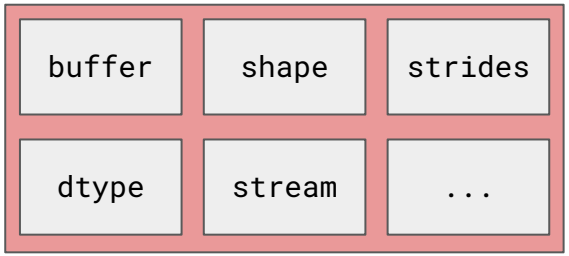
Graph-Level Escape Hatch




```
%out = "stablehlo.custom_call"(%q, %k, %v) {
 call_target_name = "flash_attn",
 has_side_effect = false,
 backend_config = {
   block_m = 128 : i32,
   block_n = 64  : i32
 },
 api_version = 4 : i32
} : (tensor<...>, tensor<...>, tensor<...>) -> tensor<...>
```

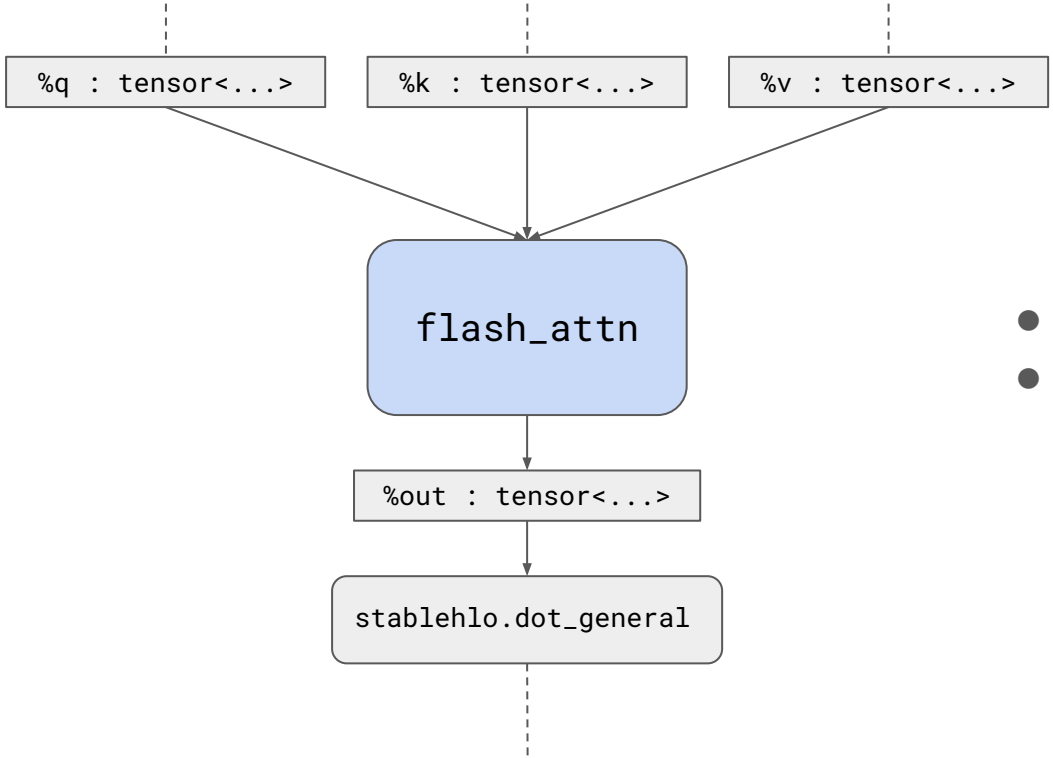
Graph-Level Escape Hatch





Mapping from logical tensor to concrete value representation is lowering-dependent

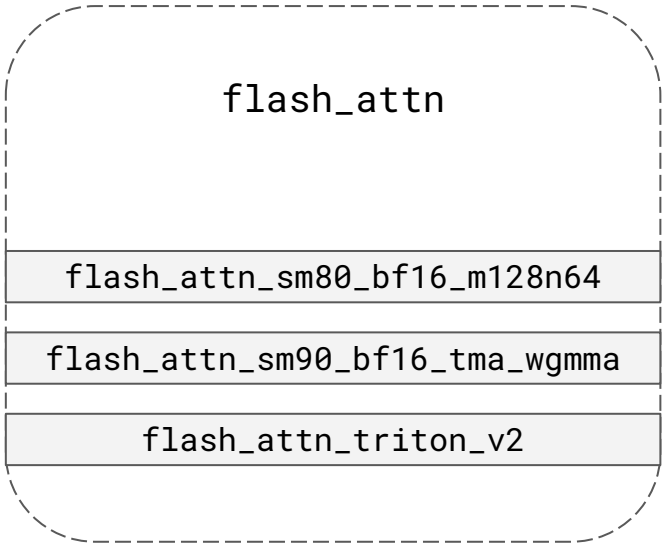
Graph-Level Escape Hatch



Invocation context: where does the foreign op run?

- Kernel launch semantics
- Workspace memory allocation

Graph-Level Escape Hatch



A foreign op is often backed by a set of many potential realizations

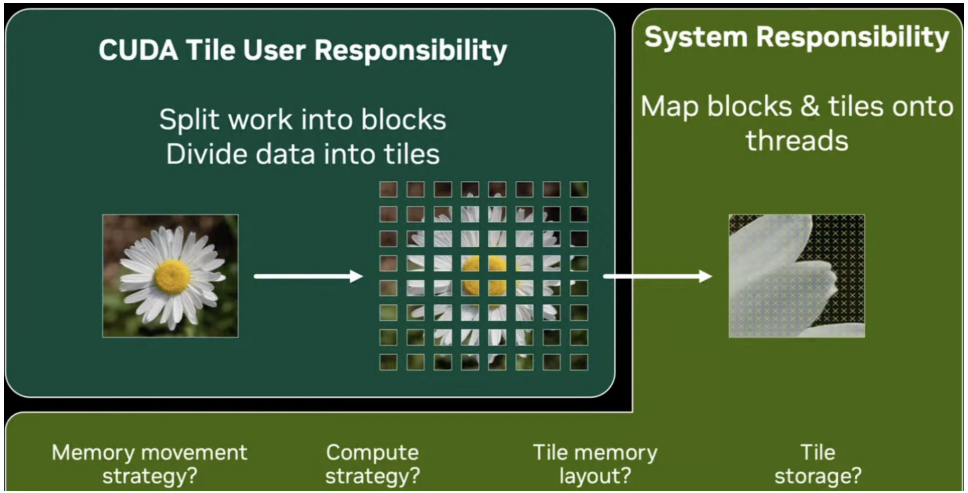
Which ones are valid?
Which ones are performant?

What The Host Must Know

Value Interface	shapes, strides, dtypes
Effects	purity, aliasing, reads/writes
Invocation Context	device/stream, launch/runtime conventions
Specialization and Dispatch	which implementation family members are legal/performant for this case

Tile Escape Hatch

- Tile-based GPU programming models make it easy to write certain types of algorithms
  - Sorting, compression, comms, FFTs, hashmaps are harder or impossible to express
- Escape hatch: call into existing SIMT code



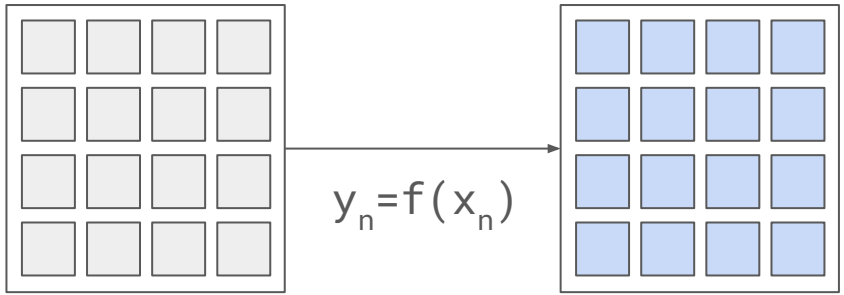
Calling SIMT from Tile

```
entry @my_kernel(...) {
 %data  = load_view_tko %input[%batch_idx]
 %out   = foreign_call @my_func(%data)
 %sq    = mulf %out, %out
 store_view_tko %sq, %output[%batch_idx]
}
```

```
__device__ void my_func(/* ... */) {
 // SIMT fragment
}
```

- Where is `data` physically stored?
- How is `data` laid out in memory?
- How many threads will call `my_func`?

Simplified Contract: Elementwise Tile Interop

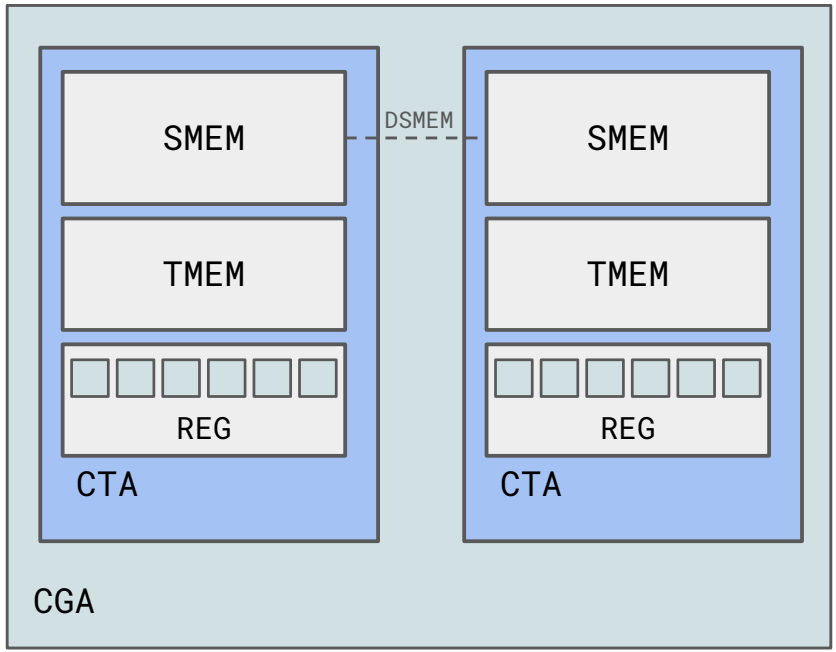



```
y = tl.inline_asm_elementwise(
    asm="ex2.approx.f32 $0,
$1;",
    constraints="=r,r",
    args=[x],
    dtype=tl.float32,
    is_pure=True,
    pack=1,
)
```

- Local value representation
- Mostly local effects

Tile Value Representation is Complex

and lowering-dependent



How is the tile distributed across threads/CTAs?

How can a SIMT function access a Tile?

```
// Interleaved layout
__device__ void my_func(float my_local_data[N]) {
 int tid = threadIdx.x;
 for (int i = 0; i < N; i++) {
     logical_coord = tid * N + i;
     my_local_data[logical_coord] = logical_coord;
 }
}

// Blocked layout
__device__ void my_func(float my_local_data[N]) {
 int tid = threadIdx.x;
 int num_threads = blockDim.x;
 for (int i = 0; i < N; i++) {
     logical_coord = num_threads * i + tid;
     my_local_data[logical_coord] = logical_coord;
 }
}
```

Invocation Context

- Number of threads is chosen by host compiler
- Shared memory allocation

```
__device__ addone_vec128_t128(float* in, float*
out) {
   // Expect to be called with 128 threads, so
each thread processes one element
   out[threadIdx.x] = in[threadIdx.x] + 1.0;
}

__device__ addone_vec128_t64(float* in, float*
out) {
   // Expect to be called with 64 threads, so
each thread processes two elements
   #pragma unroll
   for (int i = 0; i < 2; ++i) {
       int index = threadIdx.x*2 + i;
       out[index] = in[index] + 1.0;
   }
}
```

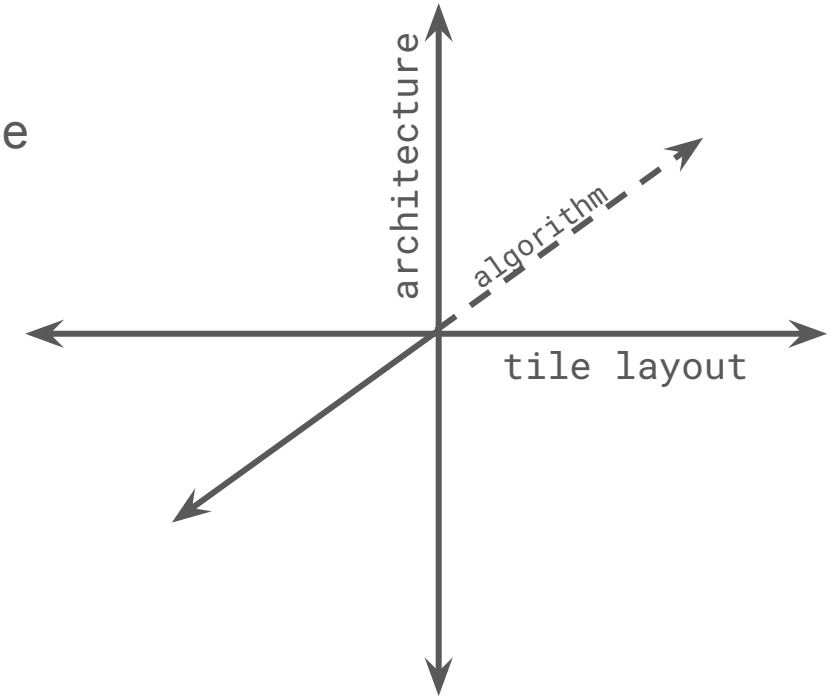
Effects / Other Considerations

- Synchronization/barriers
- Asynchrous inputs/outputs
- Warp specialized pipelining

Contract item	Graph level	Elementwise-local	Tile/SIMT
Value interface	runtime buffer/view	per-instance local value	distributed fragment
Invocation context	separate launch	local snippet	in-kernel cooperative region
Effect model	coarse effects	mostly local effects	sync / atomics / ordering
Resource model	framework-owned I/O	mostly local registers	shared kernel budget
Specialization surface	more runtime-visible	instruction / type constraints	aligned with lowering
Optimization envelope	lose fusion	mostly preserve local scheduling	lose pipelining / freedom

Multiple implementations

- Larger selection space
- Implementation selection must happen at compile time



Why The Compiler Must Do More

- Larger selection space
- Less runtime deferral
- Compile-time validation/selection/specialization

Approach A: Static Annotations

```
foreign.decl @my_addone(%tile: !tile<...>) -> !tile<...>

foreign.impl @addone_128_t128 implements @my_addone {
 ...
 num_threads = 128,
}

foreign.impl @addone_128_t64_interleaved implements @my_addone {
 ...
 num_threads = 64,
 input_layout = ...,
 output_layout = ...,
 memory_space = reg,
 execution_scope = CTA
 scratch_smem = ...
...
}
```

Approach A: Static Annotations

```
foreign.decl @my_addone(%tile: !tile<...>) -> !tile<...>

foreign.impl @addone_128_t128 implements @my_addone {
 ...
 num_threads = 128,
}

foreign.impl @addone_128_t64_interleaved implements @my_addone {
 ...
 num_threads = 64,
 input_layout = ...,
 output_layout = ...,
 memory_space = reg,
 execution_scope = CTA
 scratch_smem = ...
...
}
```

Establishes Tile-level contract

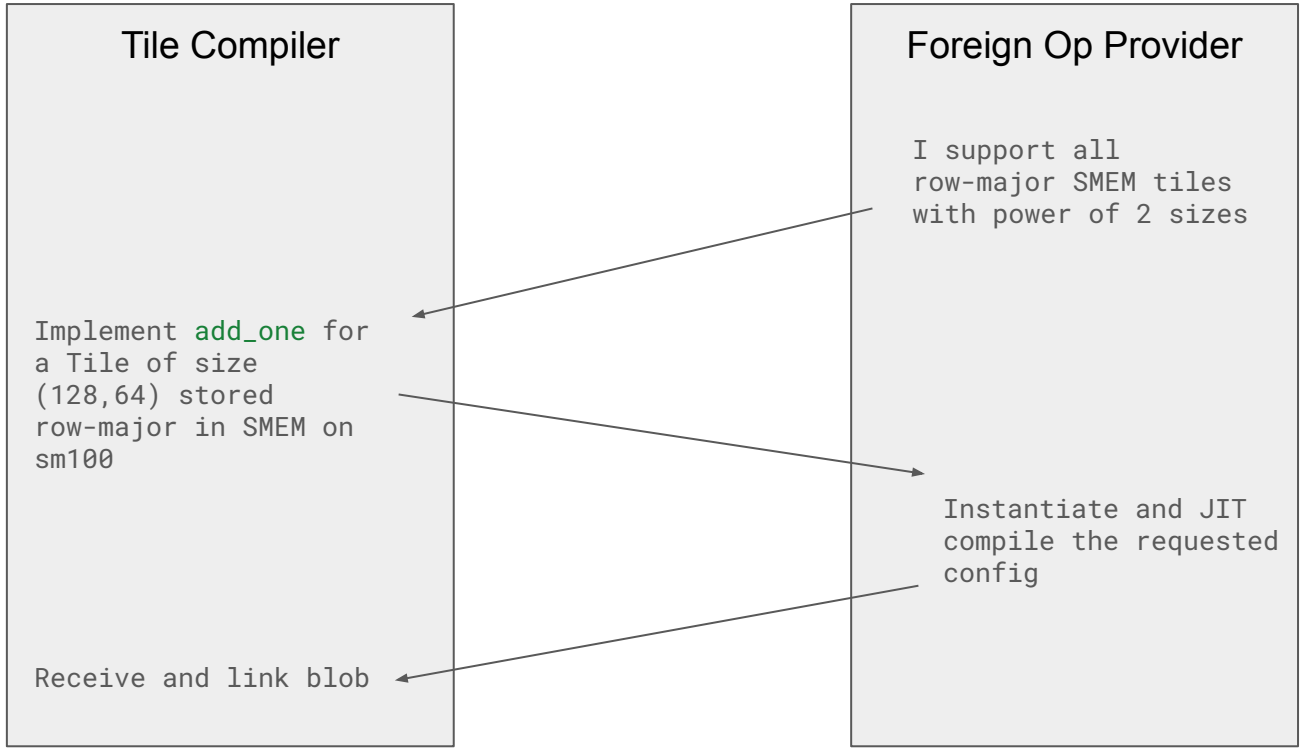
Contract for a specific implementation

Approach A: Static Annotations

```
foreign.decl @my_addone(%tile: !tile<...>) -> !tile<...>

foreign.impl @addone_128_t128 implements @my_addone {
 ...
 num_threads = 128,
}

foreign.impl @addone_128_t64_interleaved implements @my_addone {
 ...
 num_threads = 64,
 input_layout = ...,
 output_layout = ...
 ...
}

foreign.impl @addone_128_t64_blocked implements @my_addone {
 ...
 num_threads = 64,
 input_layout = ...,
 output_layout = ...
 ...
}
```

How do we know which to choose?

Approach B: Deferred Specialization



Helps with variant explosion but optimization decisions are still left up to the Tile compiler

Cost Model Composition

- Is there a meaningful way to compose the cost models for opaque operations?

```
total_cost(context, impl) != host_cost(context) + callee_cost(impl)
```

Minimal Summary or Shadow Compiler?

At what point are we just re-implementing lowering externally?

Open Questions

- What is the minimum summary required for correctness?
- What extra summary is required for performance?
- Which parts should be declarative vs. queryable?
- If foreign code wasn’t a black box and instead was in MLIR: can we make reusable interfaces to automatically infer the semantic summary?

Thank you!