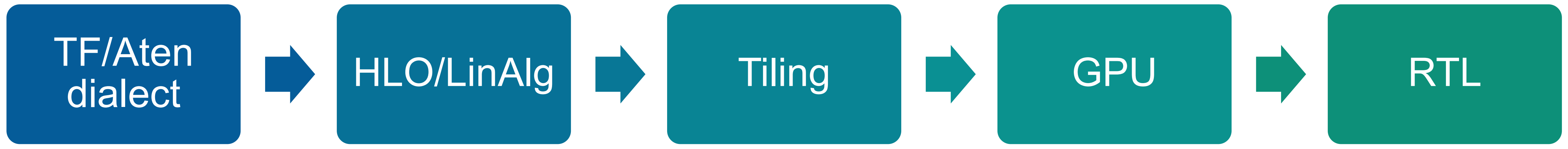
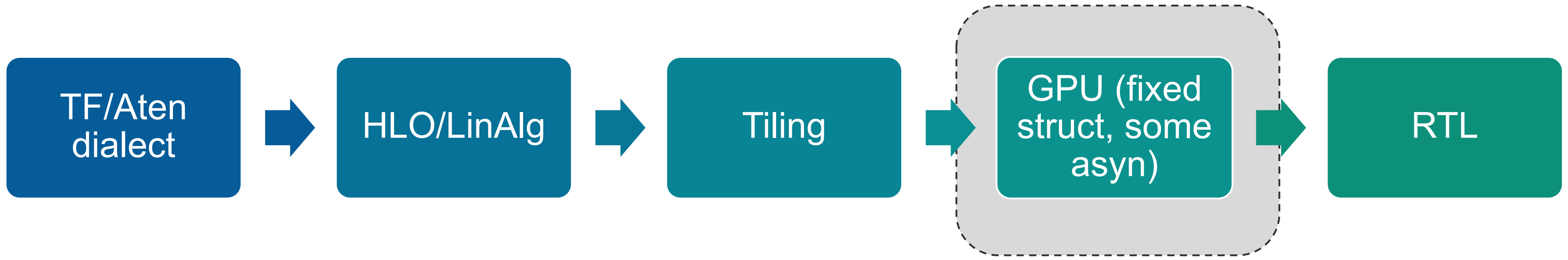
# Event Queue Dialect:
Bridging Structure and Control

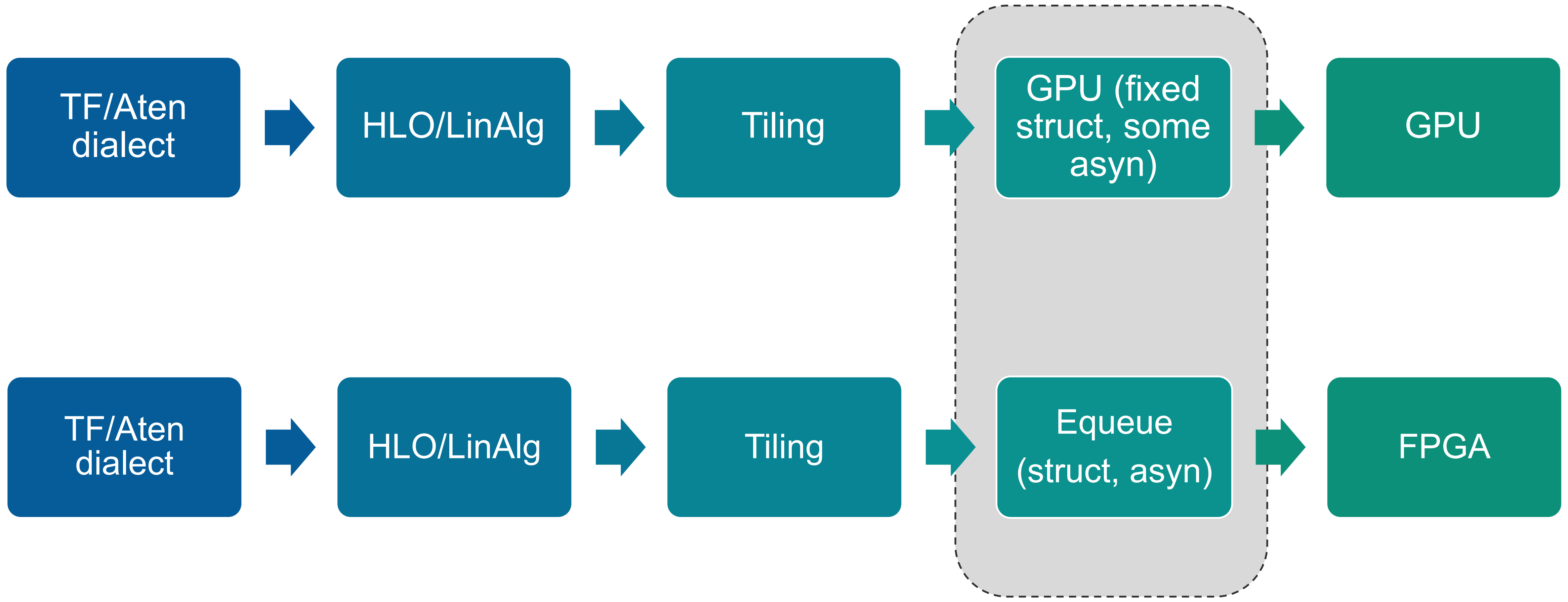
## Lowering Pipeline



## Lowering Pipeline

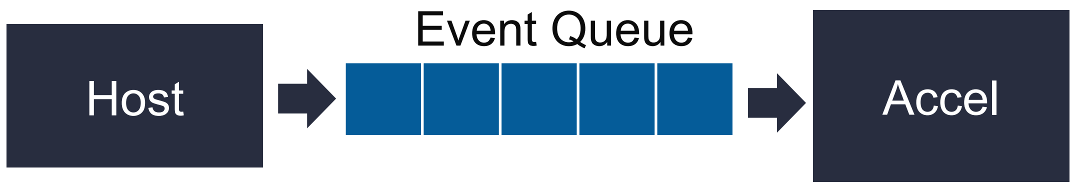


## Lowering Pipeline



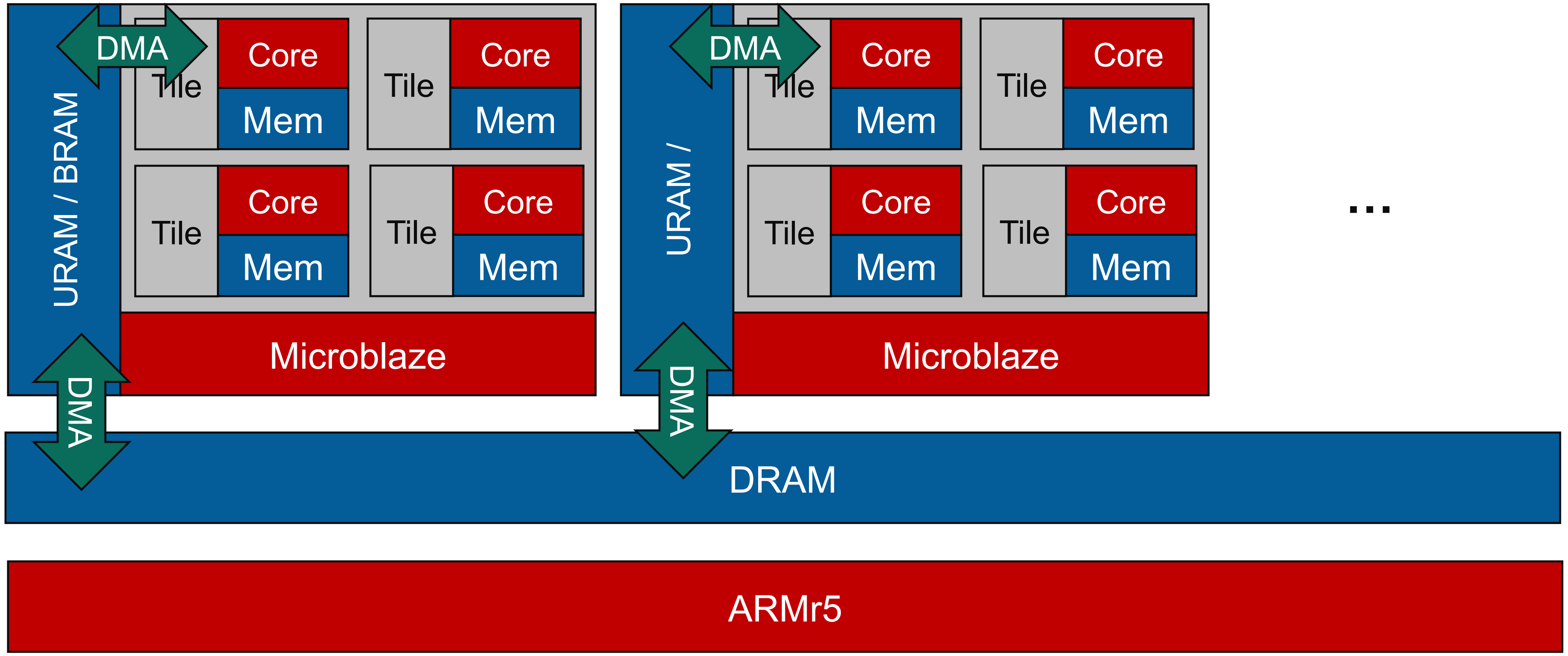
## Async Dialect

```
async.func @f(%arg0: tensor<32x8xf32>, %arg1: tensor<8x32xf32>, %arg2: tensor<8x32xf32>, %arg3: tensor<32xf32>) ->
(tensor<32x32xf32>, !async.token) {
      %0:2 = "async.execute"() ( {
            %3 = "some_computation "(%arg0, %arg1) : (tensor<32x8xf32>, tensor<8x32xf32>) -> tensor<32x32xf32>
            "async.yield"(%3) : (tensor<32x32xf32>) -> ()
      }) : () -> (!async.token, tensor<32x32xf32>)
      %1:2 = "async.execute "() ( {
            %3 = "some_computation"(%arg0, %arg1) : (tensor<32x8xf32>, tensor<8x32xf32>) -> tensor<32x32xf32>
            "async.yield"(%3) : (tensor<32x32xf32>) -> ()
      }) : () -> (!async.token, tensor<32x32xf32>)
      %2:2 = "async.execute"(%0#0, %1#0) ( {
            %3 = “some_computation”(%0#1, %1#1, %arg3) : tensor<32x32xf32>, tensor<32x32xf32>, tensor<32xf32> -> tensor<32x32xf32>
            "async.yield"(%3) : (tensor<32x32xf32>) -> ()
      }) : (!async.token, !async.token) -> (!async.token, tensor<32x32xf32>)
      "async.yield"(%2#1, %2#2) : (tensor<32x32xf32>, !async.token) -> ()
}
```



https://llvm.discourse.group/t/rfc-new-dialect-for-modelling-asynchronous-execution-at-a-higher-level/1345

# Inside Accelerator



## Motivation

- Existing dialects cannot describe programs at a detailed enough level
  - Hardware hierarchy and properties
  - Model data movement and buffer allocation
  - Heterogenous processors with distributed event-based control

## Motivation

- Existing dialects cannot describe programs at a detailed enough level
  - Hardware hierarchy and properties
  - Model data movement and buffer allocation
  - Heterogenous processors with distributed event-based control

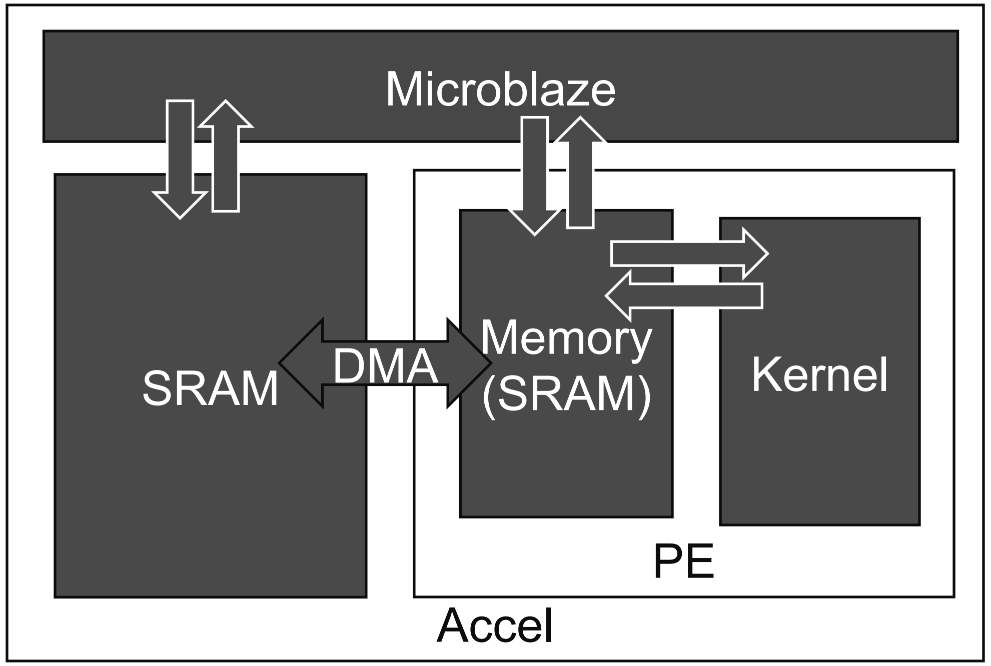
- Accurate performance estimation of heterogenous system
  - High abstraction level
  - Fast estimation
  - Custom lowering

## High Level Dialect – PyTorch to ATen

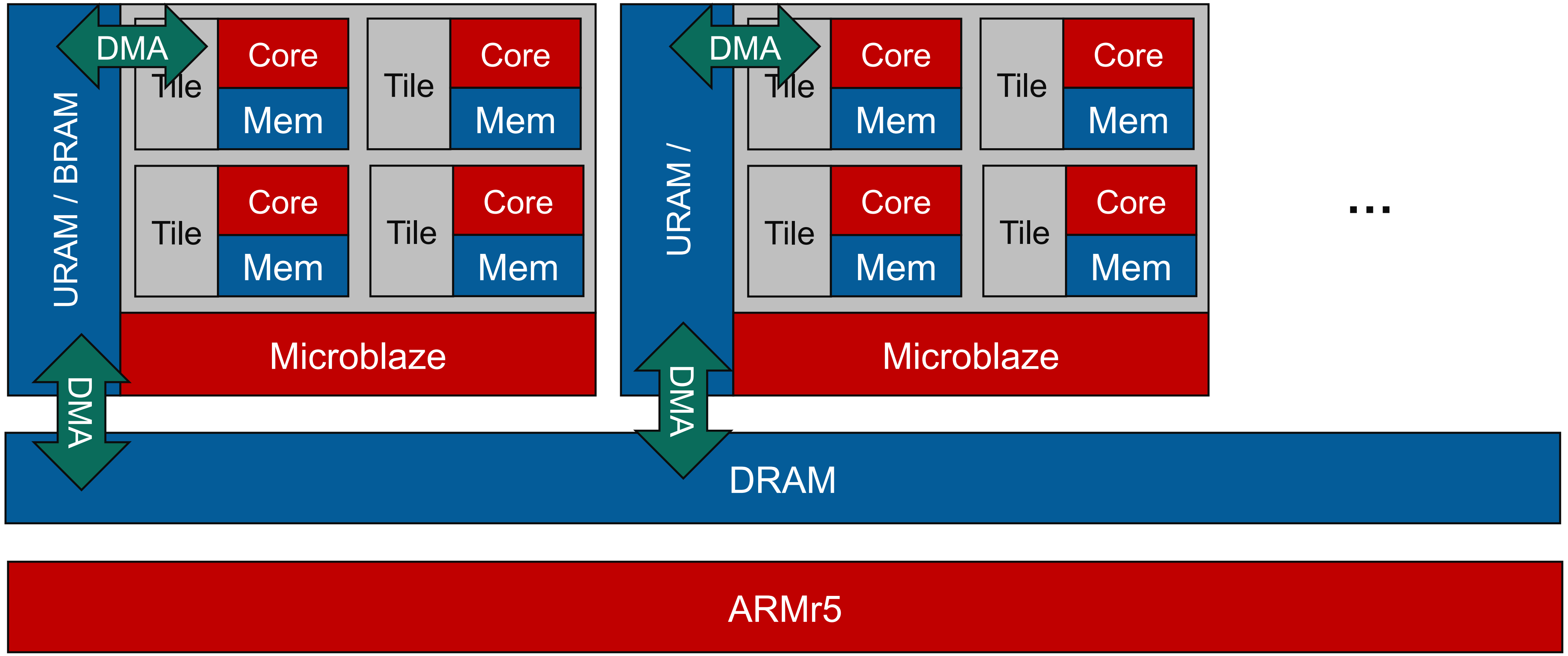
```
func @graph( %arg0: tensor<16xf32>, %arg1: tensor<16xf32>) -> tensor<16xf32> {
        %0 = aten.constant()
        %arg2 = aten.add(%arg0, %arg1, %0) {acdc_layer_name = “L0-add-0”}
        return %arg2
}
```

## High Level Dialect – Extending ATen

```
func @graph( %arg0: memref<16xf32>, %arg1:
             memref<16xf32>) -> memref<16xf32> {

   %m0 = aten.acap_alloc()
   %s0 = aten.acap_copy(%m0, %arg0)
   %v0 = aten.acap_tensor_load(%m0, %s0)

   %m1 = aten.acap_alloc()
   %s1 = aten.acap_copy(%m1, %arg0)
   %v1 = aten.acap_tensor_load(%m1, %s1)

   %s2 = aten.constant()
   %m2 = aten.acap_alloc()
   %v2 = aten.add(%v0, %v1, %s2)
   %s3 = aten.acap_tensor_store(%v2, %m2)
   %m3 = aten.acap_alloc()
   %s4 = aten.acap_copy(%m2, %m3, %s3)
   %s5 = aten.acap_wait_all(%s4)
   return %m3
}
```



# Inside Accelerator



## EQueue Dialect

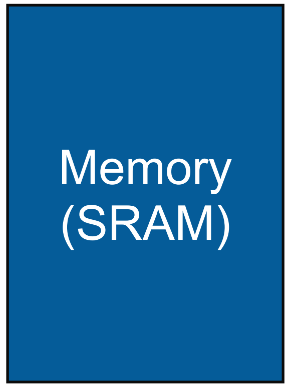
- Model structure and capture hardware properties
- Explicit data movements
- Concurrency between controllers
- Simulator

- Case study on input stationary dataflow on systolic array
- Different levels of abstractions from linalg to equeue dialect

## Modeling Hardware

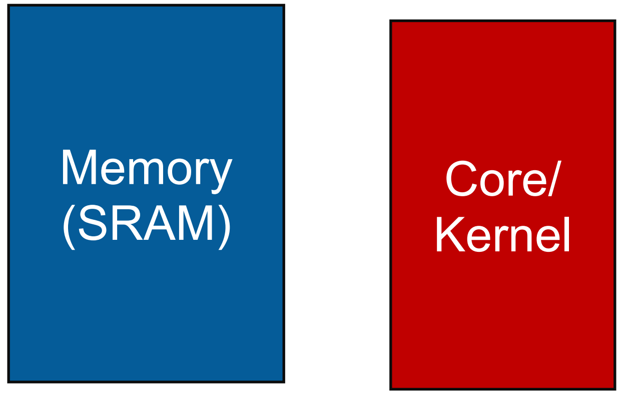
## Modeling Hardware

```
%aie_mem = equeue.create_mem  “mem”, [11], f32, SRAM
```



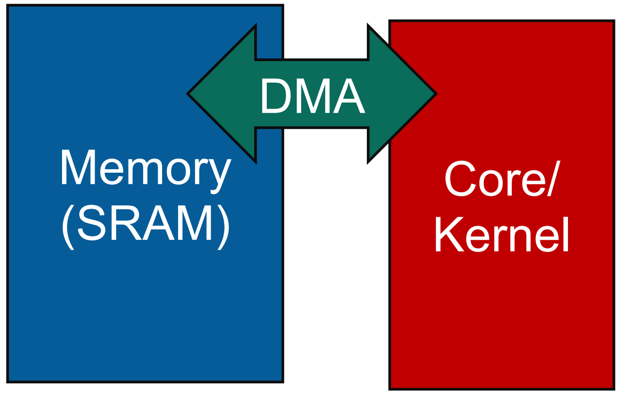
## Modeling Hardware

```
%aie_mem = equeue.create_mem “mem”, [11], f32, SRAM

%aie_core = equeue.create_proc “core”, AIEngine
```



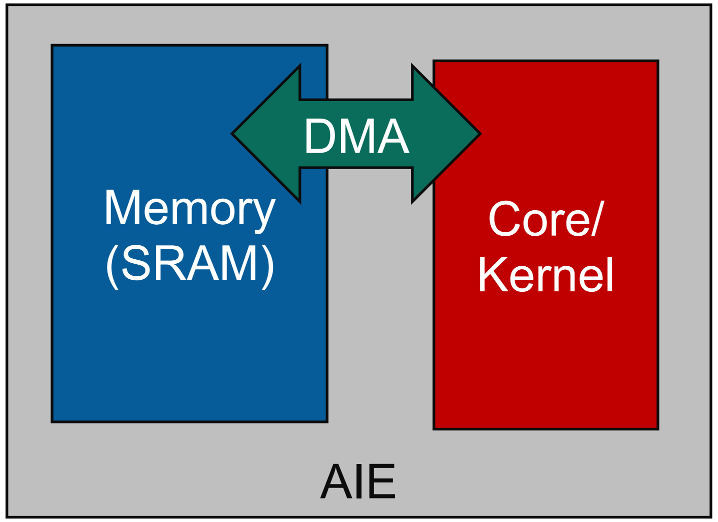
## Modeling Hardware

```
%aie_mem = equeue.create_mem "mem", [11], f32, SRAM

%aie_core = equeue.create_proc "core", AIEngine

%dma = equeue.create_dma "dma":()->i32
```



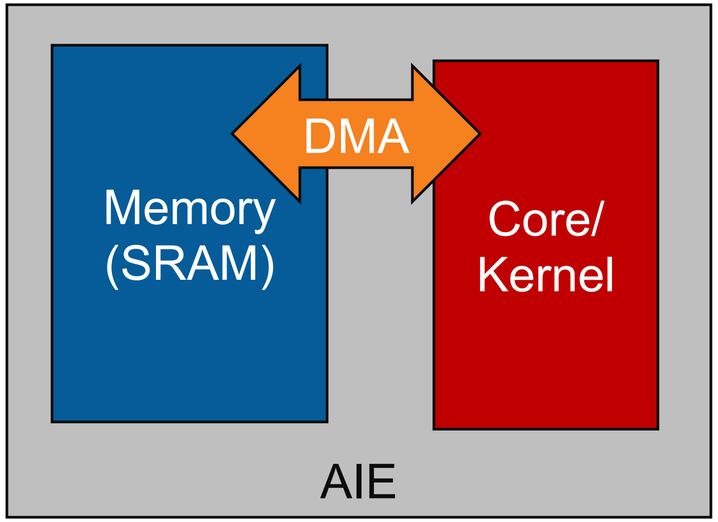
# Arbitrary Hardware Hierarchy

```
%aie_mem = equeue.create_mem “mem”, [11], f32, SRAM

%aie_core = equeue.create_proc “core”, AIEngine

%dma = equeue.create_dma “dma”:()->i32

%aie = equeue.create_comp (“aie”, %aie_mem, %aie_core, %dma) :
                                                (i32, i32, i32)->i32
```



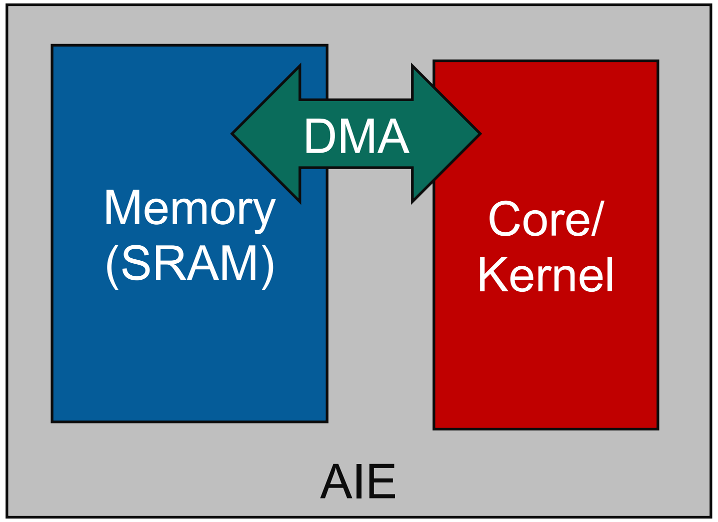
# Arbitrary Hardware Hierarchy

```
%aie_mem = equeue.create_mem "mem", [11], f32, SRAM

%aie_core = equeue.create_proc "core", AIEngine

%dma = equeue.create_dma "dma":()->i32

%aie = equeue.create_comp ("aie", %aie_mem, %aie_core, %dma) :
                                                          (i32, i32, i32)->i32
%dma = equeue.get_comp %aie, "dma"
```

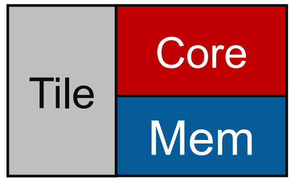


# Arbitrary Hardware Hierarchy with FPGA Property

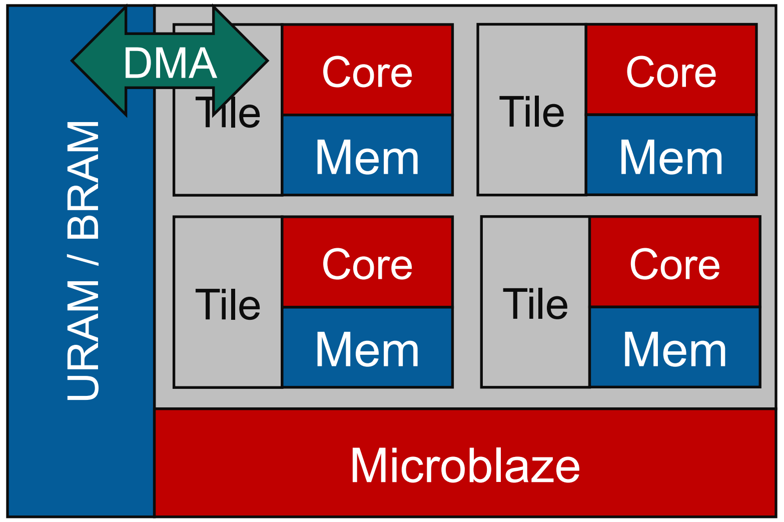
```
%aie_mem = equeue.create_mem “mem”, [11], f32, SRAM

%aie_core = equeue.create_proc “core”, AIEngine

%dma = equeue.create_dma “dma”:()->i32

%aie = equeue.create_comp (“aie”, %aie_mem, %aie_core, %dma) :
                                              (i32, i32, i32)->i32
```



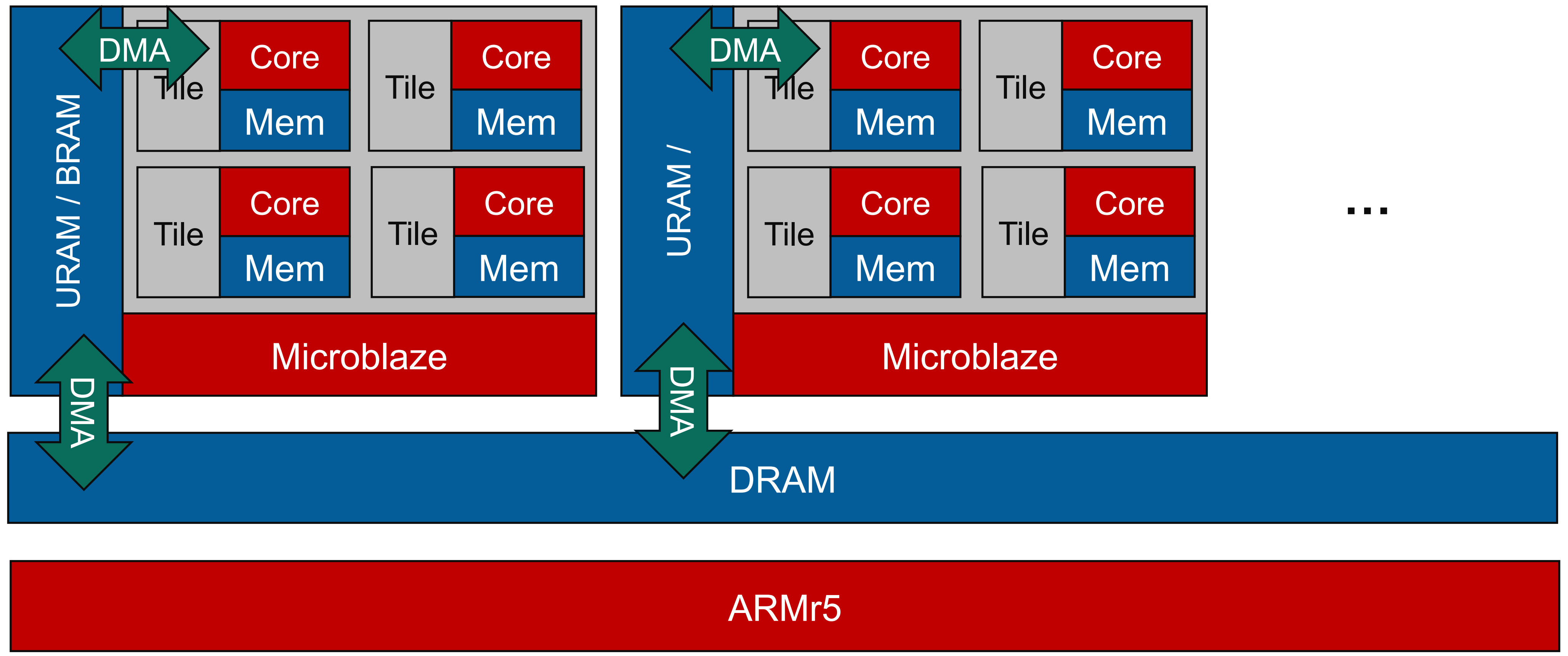
## Modeling Large Scale Hardware



# Modeling Large Scale Hardware



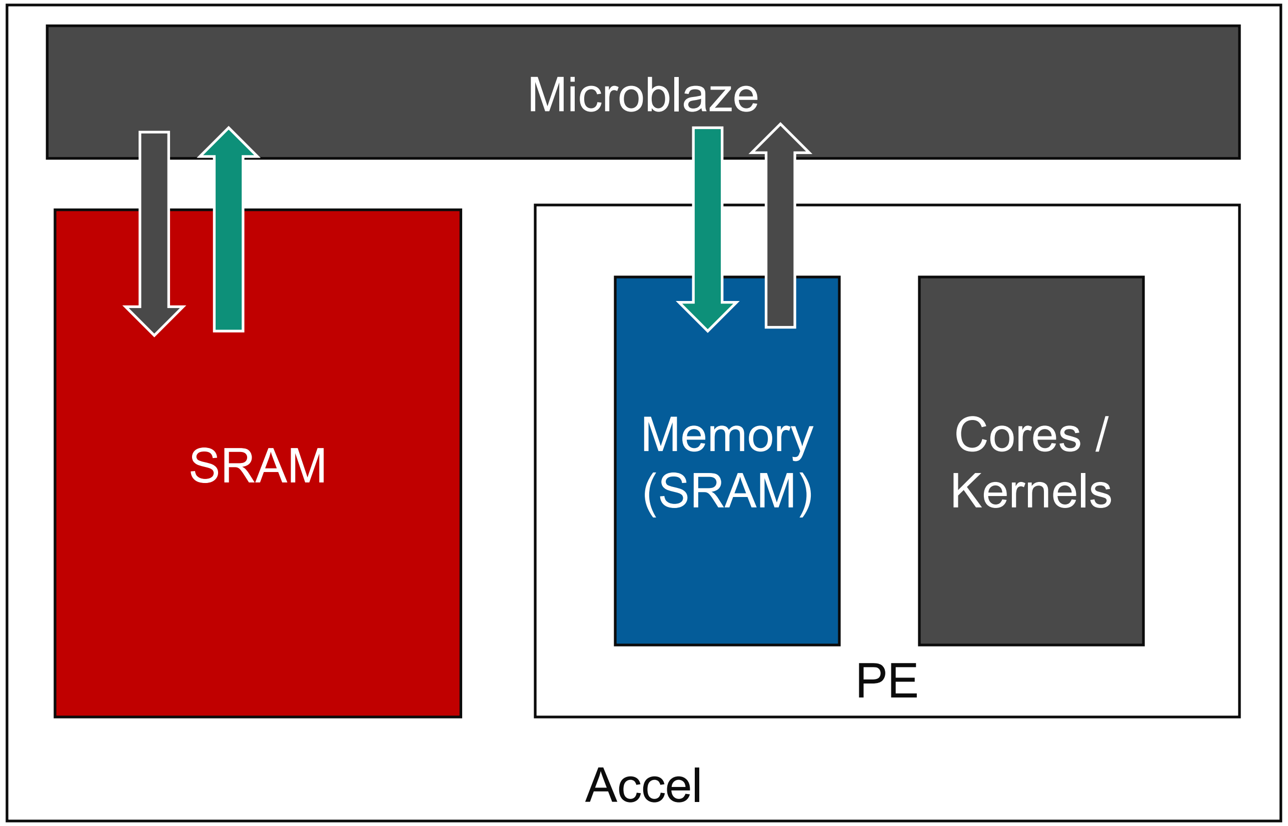
# Modeling Large Scale Hardware



## Explicit Data Movements

# Explicit Data Movements among Memory

```
%sram = equeue.create_mem “sram”, [64], f32, SRAM

%mem = equeue.create_mem “mem”, [4], f32, SRAM


%sram_buffer = equeue.alloc %sram, [4], f32

%pe_buffer = equeue.alloc %mem, [4], f32


%data = equeue.read( %sram_buffer )

equeue.write( %data, %pe_buffer )


equeue.dealloc(%sram_buffer)

equeue.dealloc(%pe_buffer)
```

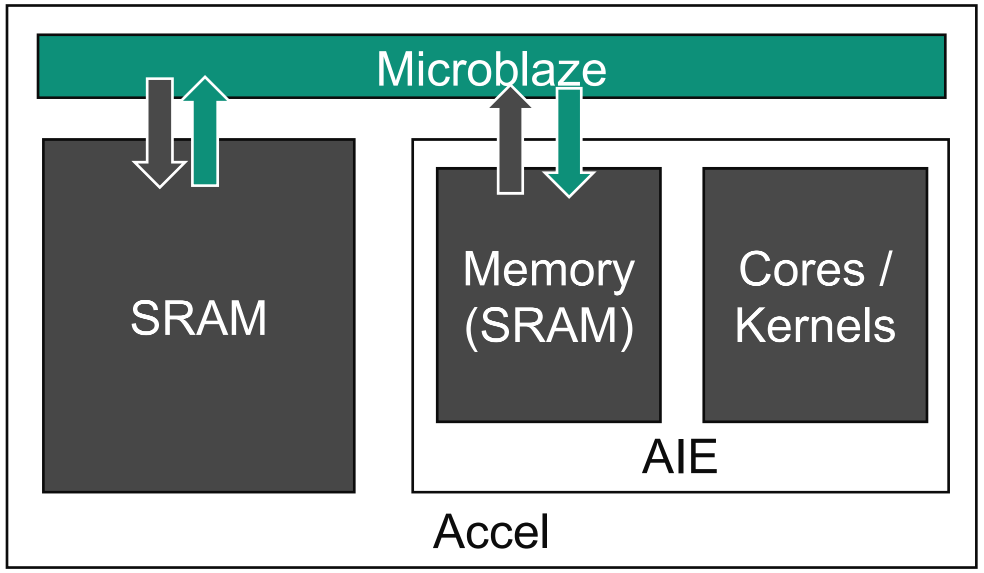


## Concurrency between Controllers

## Concurrency

- A processor can issue and receive launch operation

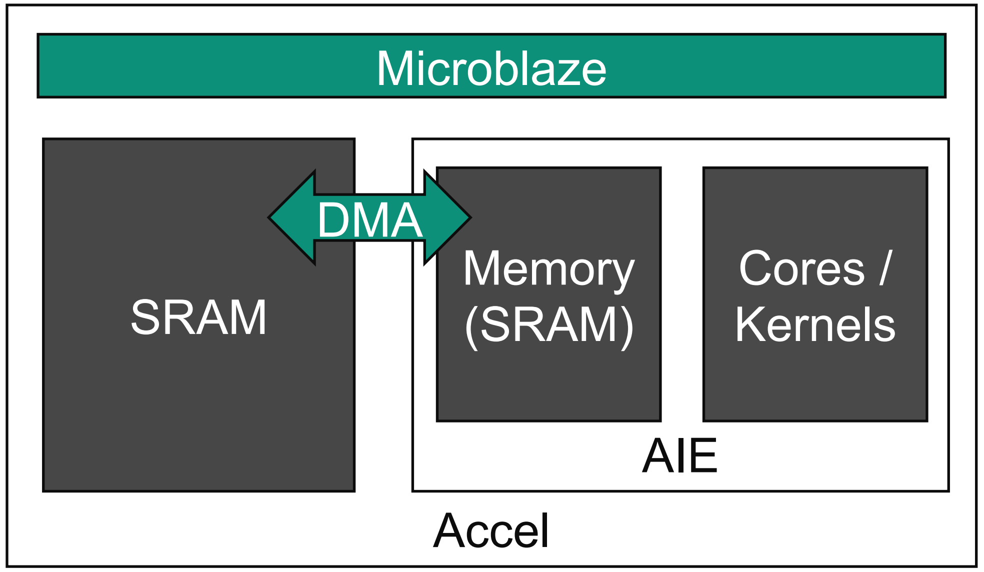
```
%microblaze = equeue.create_proc Microblaze
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    %data = equeue.read(%sram_buffer)
    equeue.write(%data, %aie_buffer)
    …
    equeue.return
}
```



## Concurrency

- A processor can issue and receive launch operation

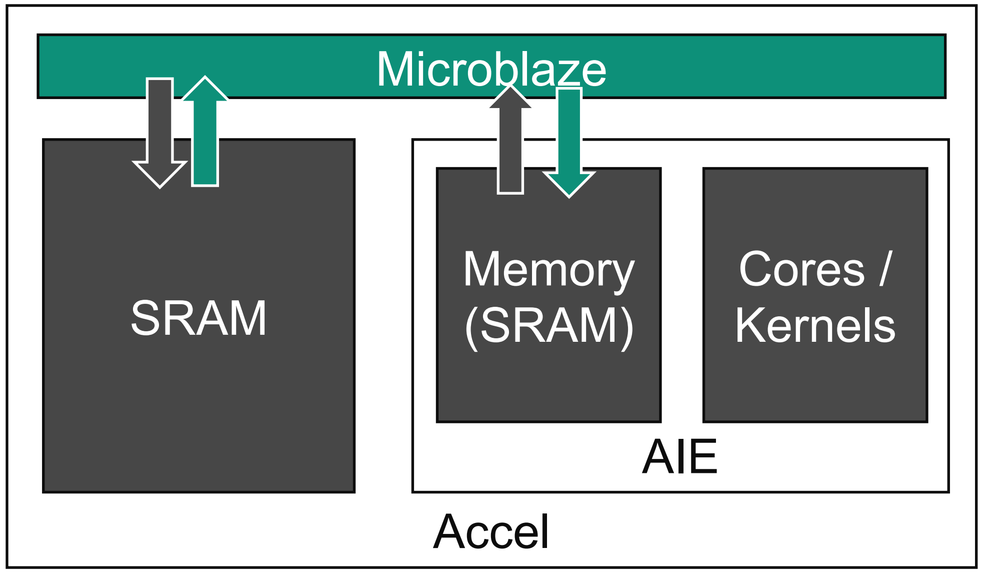
```
%microblaze = equeue.create_proc Microblaze
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    …
    %done_dma = equeue.memcpy(%start_dma, %sram_buffer, %aie_buffer,
    %dma)
    …
    equeue.return

}
```



## Concurrency

- A processor can issue and receive launch operation
  - For launching operation, all resources requires explicit handler to pass control.

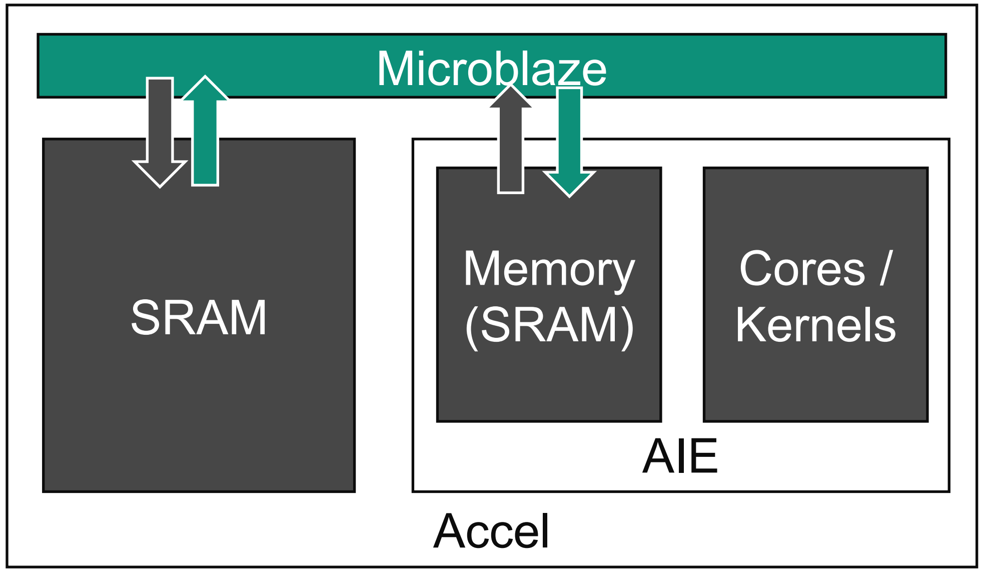
```
%microblaze = equeue.create_proc Microblaze
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    …
    %done_dma = equeue.memcpy(%start_dma, %sram_buffer, %aie_buffer,
    %dma)
    …
    equeue.return

}
```



# Concurrency

- A processor can issue and receive launch operation
  - For launching operation, all resources requires explicit handler to pass control.
  - Processors are asynchronous: starts on receiving the start event ends with a done event.

```
%microblaze = equeue.create_proc Microblaze
%start = memcpy(%start_memcpy, …)
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    …
    %done_dma = equeue.memcpy(%start_dma, %sram_buffer, %aie_buffer,
    %dma)
    …
    equeue.return
}
```



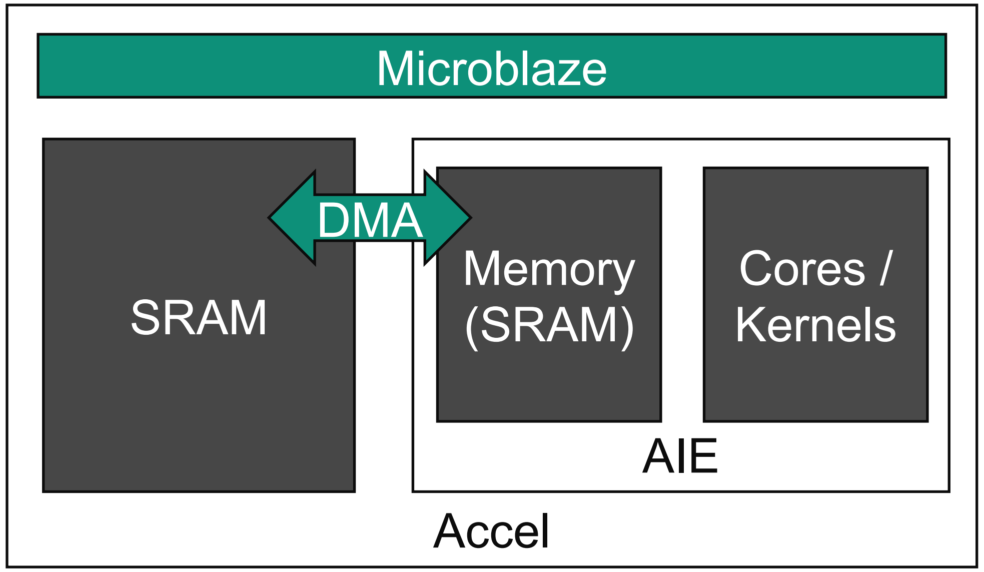
## Concurrency

- A processor can issue and receive launch operation
  - For launching operation, all resources requires explicit handler to pass control.
  - Processors are asynchronous: starts on receiving the start event ends with a done event.
  - Event queue: stores operation waiting for events

```
%microblaze = equeue.create_proc Microblaze
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    …
    %done_dma = equeue.memcpy(%start_dma, %sram_buffer, %aie_buffer,
    %dma)
    …
    equeue.return
}
```

| DMA |
|---|
| memcpy |

| Microblaze |
|---|
| launch |

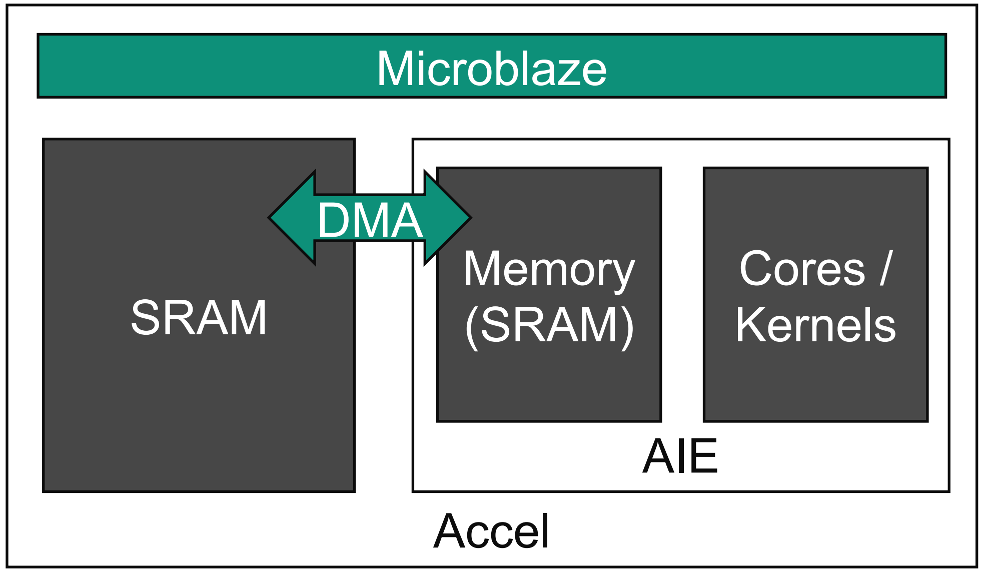


# Concurrency

- A processor can issue and receive launch operation
  - For launching operation, all resources requires explicit handler to pass control.
  - Processors are asynchronous: starts on receiving the start event ends with a done event.
  - Event queue: stores operation waiting for events
  - Sequential Execution

| DMA |
|---|
| |

```
%microblaze = equeue.create_proc Microblaze
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    …
    %done_dma = equeue.memcpy(%start_dma, %sram_buffer, %aie_buffer,
    %dma)
    …
    equeue.return
}
```

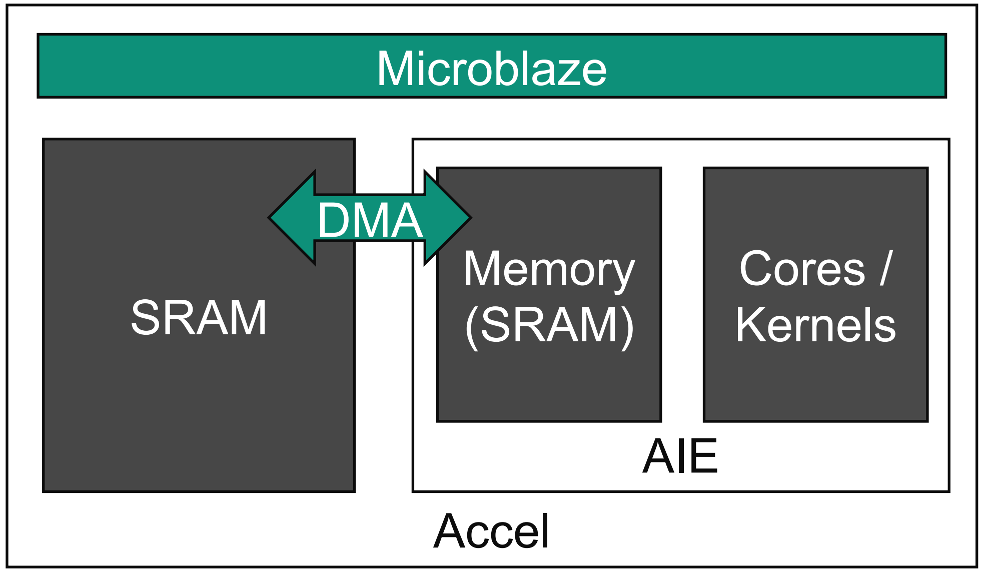


## Concurrency

- A processor can issue and receive launch operation
  - For launching operation, all resources requires explicit handler to pass control.
  - Processors are asynchronous: starts on receiving the start event ends with a done event.
  - Event queue: stores operation waiting for events
  - Sequential Execution

| DMA |
|---|
| |

```
%microblaze = equeue.create_proc Microblaze
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    …
    %done_dma = equeue.memcpy(%start_dma, %sram_buffer, %aie_buffer,
    %dma)
    …
    equeue.return
}
```

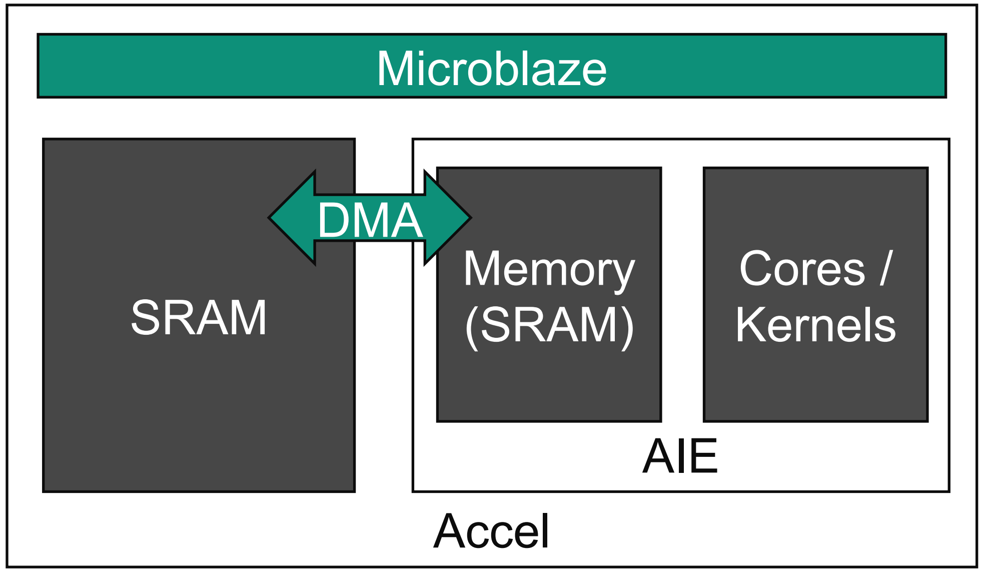


# Concurrency

- A processor can issue and receive launch operation
  - For launching operation, all resources requires explicit handler to pass control.
  - Processors are asynchronous: starts on receiving the start event ends with a done event.
  - Event queue: stores operation waiting for events
  - Sequential Execution

| DMA |
|---|
| memcpy |

```
%microblaze = equeue.create_proc Microblaze
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    …
    %done_dma = equeue.memcpy(%start_dma, %sram_buffer, %aie_buffer,
    %dma)
    …
    equeue.return
}
```

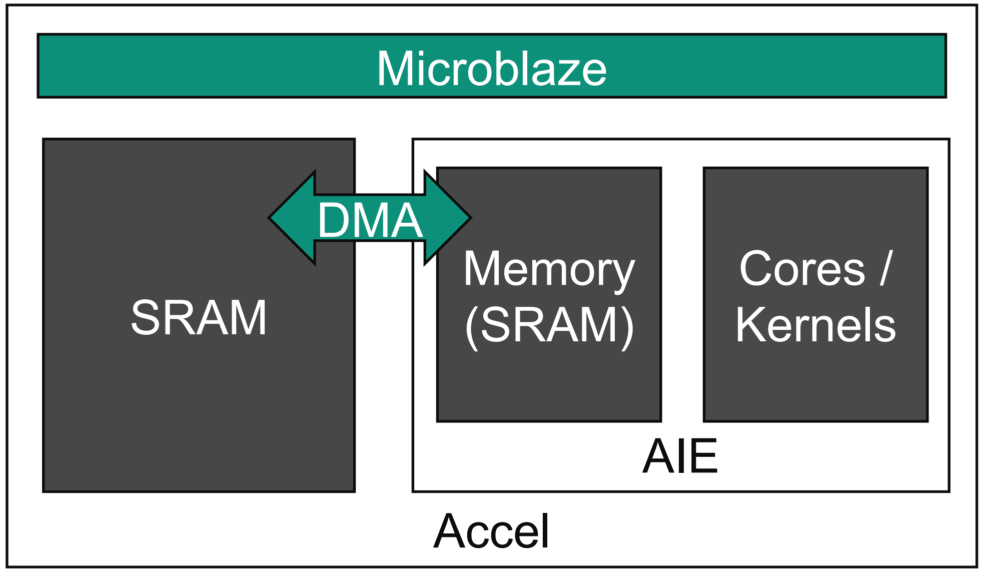


## Concurrency

- A processor can issue and receive launch operation
  - For launching operation, all resources requires explicit handler to pass control.
  - Processors are asynchronous: starts on receiving the start event ends with a done event.
  - Event queue: stores operation waiting for events
  - Sequential Execution

| DMA |
|---|
| memcpy |

```
%microblaze = equeue.create_proc Microblaze
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    …
    %done_dma = equeue.memcpy(%start_dma, %sram_buffer, %aie_buffer,
    %dma)
  ➡ …
    equeue.return
}
```

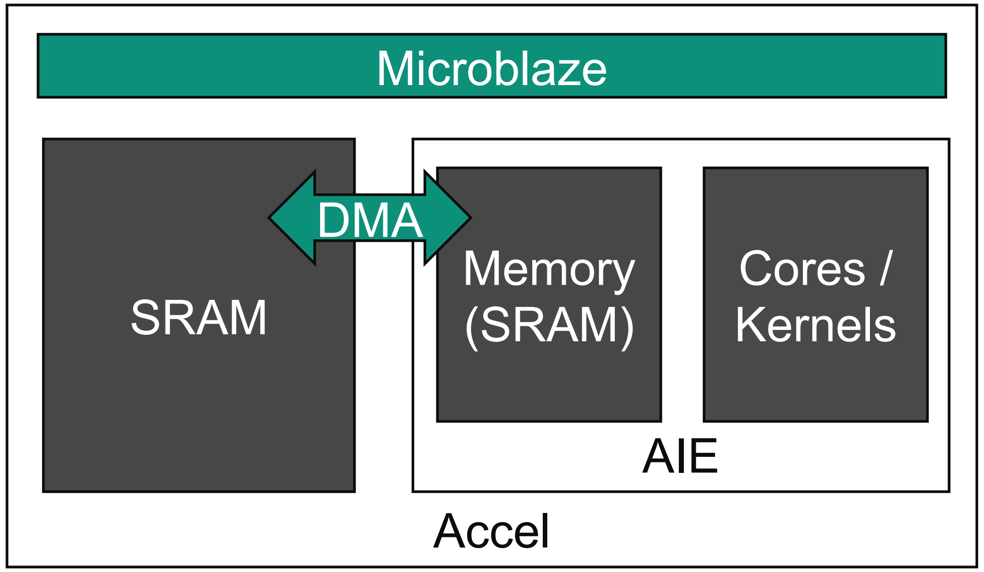


# Concurrency

- A processor can issue and receive launch operation
  - For launching operation, all resources requires explicit handler to pass control.
  - Processors are asynchronous: starts on receiving the start event ends with a done event.
  - Event queue: stores operation waiting for events
  - Sequential Execution

| DMA |
|---|
| memcpy |

```
%microblaze = equeue.create_proc Microblaze
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    …
    %done_dma = equeue.memcpy(%start_dma, %sram_buffer, %aie_buffer,
    %dma)
    …
➡   equeue.return
}
```

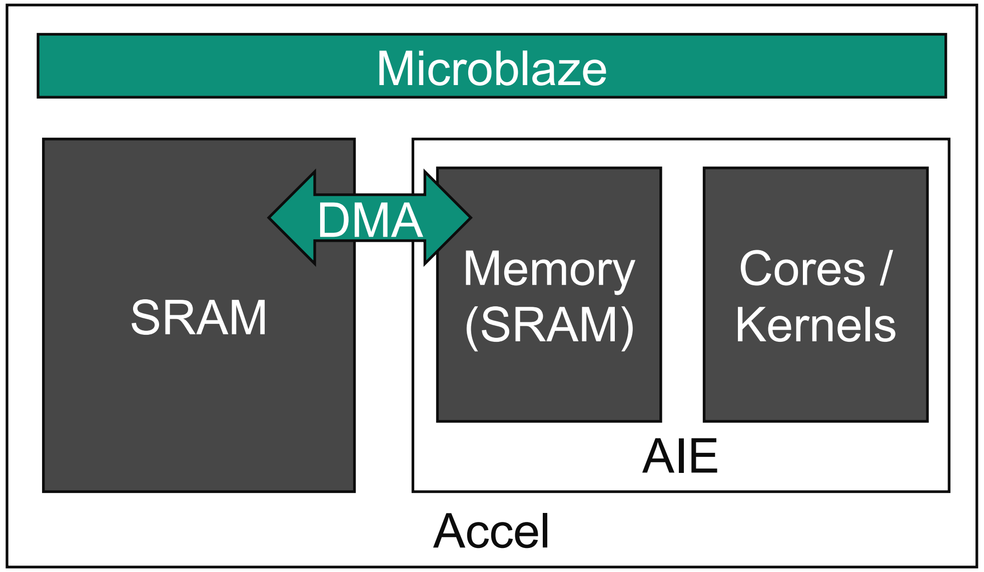


## Concurrency

- A processor can issue and receive launch operation
  - For launching operation, all resources requires explicit handler to pass control.
  - Processors are asynchronous: starts on receiving the start event ends with a done event.
  - Event queue: stores operation waiting for events
  - equeue.await: block till events happen

| DMA |
|---|
| |

```
%microblaze = equeue.create_proc Microblaze
%done = equeue.launch (%sram, %mem, %dma
= %0, %1, %2) in ( %start, %microblaze){
    %sram_buffer = equeue.alloc %sram, [4], f32
    %aie_buffer = equeue.alloc %mem, [4], f32
    …
    %done_dma = equeue.memcpy(%sram_buffer, %aie_buffer, %start_dma)
    …
➡   equeue.await(%done_dma)
    equeue.return

}
```



## A Complete Flow

## Rewriting Aten Dialect

```
func @graph( %arg0: memref<16xf32>, %arg1:
             memref<16xf32>) -> memref<16xf32> {

   %m0 = aten.acap_alloc()
   %s0 = aten.acap_copy(%m0, %arg0)
   %v0 = aten.acap_tensor_load(%m0, %s0)

   %m1 = aten.acap_alloc()
   %s1 = aten.acap_copy(%m1, %arg0)
   %v1 = aten.acap_tensor_load(%m1, %s1)

   %s2 = aten.constant()
   %m2 = aten.acap_alloc()
   %v2 = aten.add(%v0, %v1, %s2)
   %s3 = aten.acap_tensor_store(%v2, %m2)
   %m3 = aten.acap_alloc()
   %s4 = aten.acap_copy(%m2, %m3, %s3)
   %s5 = aten.acap_wait_all(%s4)
   return %m3
}
```

```
equeue.launch(%kernel, %mem, %dma, %arg0, %arg1, %arg2 =
%aie, %SRAM, %DMA, %0, %1, %2) in (%s2, %microblaze){
     %start = equeue.control_start ()
     %m0 = equeue.alloc %mem, [16], f32
     %m1 = equeue.alloc %mem, [16], f32
     %m2 = equeue.alloc %mem, [16], f32
     %s0 = equeue.memcpy(%start, %arg0, %m0, %dma)
     %s1 = equeue.memcpy(%start, %arg1, %m1, %dma)
     %s2 = equeue.control_and(%s0, %s1)
     %s3= equeue.launch(%k0, %k1, %k2 = %m0, %m1, %m2) in
     (%s2, %kernel) {
          %v0 = equeue.read(%k0)
          %v1 = equeue.read(%k1)
          %v2 = addf(%v1, %v2)
          equeue.write(%v2, %m2)
          equeue.return
     }
     %s4 = equeue.memcpy(%s3, %arg2, %2, %dma)
     %s5 = equeue.await(%s4)
     equeue.return}
```

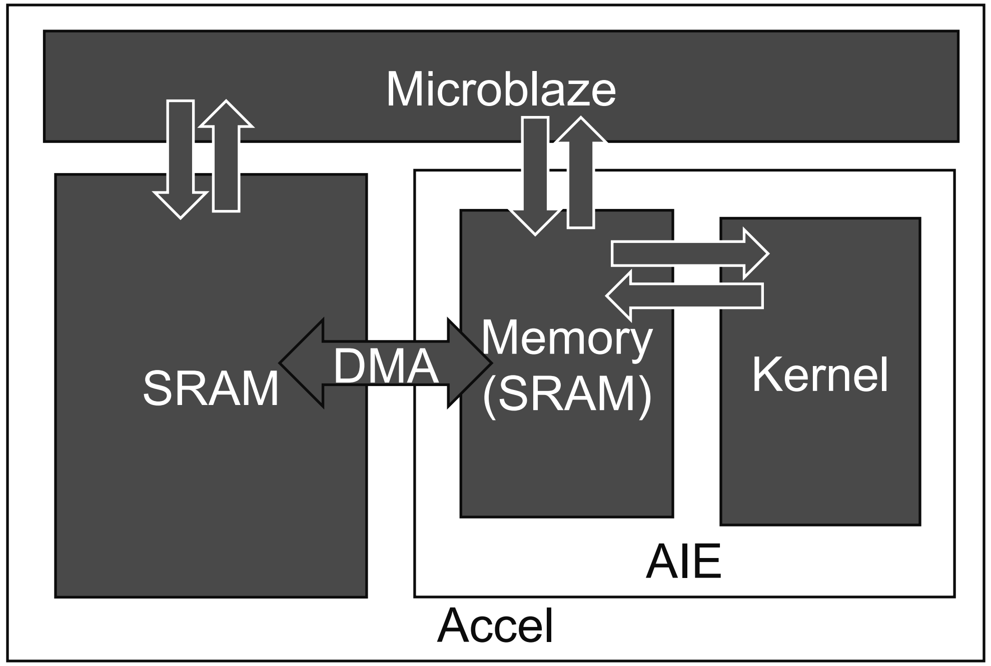
# Rewriting Aten Dialect

```
func @graph( %arg0: memref<16xf32>, %arg1:
            memref<16xf32>) -> memref<16xf32> {

   %m0 = aten.acap_alloc()
   %s0 = aten.acap_copy(%m0, %arg0)
   %v0 = aten.acap_tensor_load(%m0, %s0)

   %m1 = aten.acap_alloc()
   %s1 = aten.acap_copy(%m1, %arg0)
   %v1 = aten.acap_tensor_load(%m1, %s1)

   %s2 = aten.constant()
   %m2 = aten.acap_alloc()
   %v2 = aten.add(%v0, %v1, %s2)
   %s3 = aten.acap_tensor_store(%v2, %m2)
   %m3 = aten.acap_alloc()
   %s4 = aten.acap_copy(%m2, %m3, %s3)
   %s5 = aten.acap_wait_all(%s4)
   return %m3
}
```

```
equeue.launch(%kernel, %mem, %dma, %arg0, %arg1, %arg2 =
%aie, %SRAM, %DMA, %0, %1, %2) in (%s2, %microblaze){
      %start = equeue.control_start ()
      %m0 = equeue.alloc %mem, [16], f32
      %m1 = equeue.alloc %mem, [16], f32
      %m2 = equeue.alloc %mem, [16], f32
      %s0 = equeue.memcpy(%start, %arg0, %m0, %dma)
      %s1 = equeue.memcpy(%start, %arg1, %m1, %dma)
      %s2 = equeue.control_and(%s0, %s1)
      %s3= equeue.launch(%k0, %k1, %k2 = %m0, %m1, %m2) in
      (%s2, %kernel) {
            %v0 = equeue.read(%k0)
            %v1 = equeue.read(%k1)
            %v2 = addf(%v1, %v2)
            equeue.write(%v2, %m2)
            equeue.return
      }
      %s4 = equeue.memcpy(%s3, %arg2, %2, %dma)
      %s5 = equeue.await(%s4)
      equeue.return}
```

```
%SRAM = create_mem
[1024], f32, SRAM

launch(%mem = %SRAM)
```

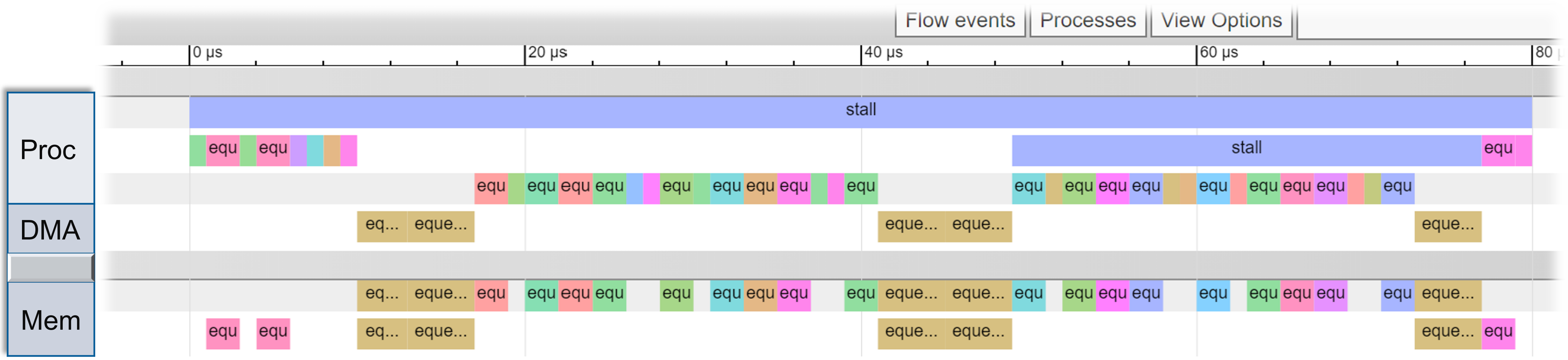
# Rewriting Aten Dialect



```
equeue.launch(%kernel, %mem, %dma, %arg0, %arg1, %arg2 =
%aie, %SRAM, %DMA, %0, %1, %2) in (%s2, %microblaze){
        %start = equeue.control_start ()
        %m0 = equeue.alloc %mem, [16], f32
        %m1 = equeue.alloc %mem, [16], f32
        %m2 = equeue.alloc %mem, [16], f32
        %s0 = equeue.memcpy(%start, %arg0, %1, %dma)
        %s1 = equeue.memcpy(%start, %arg1, %4, %dma)
        %s2 = equeue.control_and(%s0, %s1)
        %s3= equeue.launch(%k0, %k1, %k2 = %m0, %m1, %m2) in
        (%s2, %kernel) {
                %v0 = equeue.read(%k0)
                %v1 = equeue.read(%k1)
                %v2 = addf(%v1, %v2)
                equeue.write(%v2, %m2)
                equeue.return
        }
        %s4 = equeue.memcpy(%s3, %arg2, %2, %dma)
        %s5 = equeue.await(%s4)
        equeue.return}
```

```
%start_signal = "equeue.control_start"():()->!equeue.signal
%done_signal, %result = equeue.launch (%act, %weight, %aie_core_local, %dma, %m0, %m1 = %act_in, %weight_in,
%aie_core, %accel_dma, %sram, %aie_mem : tensor<16xf32>, tensor<5xf32>, i32, i32, i32, i32)
in (%start_signal, %accel_core) : tensor<12xf32> {
        %weight_sram = equeue.alloc %m0, [5], f32 : !equeue.container<tensor<5xf32>, i32>
        "equeue.write"(%weight, %weight_sram): (tensor<5xf32>, !equeue.container<tensor<5xf32>, i32>)->()
        %act_sram = equeue.alloc %m0, [16], f32: !equeue.container<tensor<16xf32>, i32>
        "equeue.write"(%act, %act_sram): (tensor<16xf32>, !equeue.container<tensor<16xf32>, i32>)->()
        %output_sram = equeue.alloc %m0, [16], f32 : !equeue.container<tensor<12xf32>, i32>
        %weight_mem = equeue.alloc %m1, [5], f32:!equeue.container<tensor<5xf32>, i32>
        %act_mem = equeue.alloc %m1, [5], f32 :!equeue.container<tensor<5xf32>, i32>
        %ofmap_mem = equeue.alloc %m1, [1], f32 :!equeue.container<f32, i32>
        %start_memcpy = "equeue.control_start"():()->!equeue.signal
        %dma_done0 = "equeue.memcpy"(%start_memcpy, %weight_sram, %weight_mem, %dma): (!equeue.signal, !equeue.container<tensor<5xf32>,i32>, !equeue.container<tensor<5xf32>,i32>, i32) -> !equeue.signal
        %c0 = constant 0:index
        %c12 = constant 2:index
        %c1 = constant 1:index
        %compute_done = scf.for %k = %c0 to %c12 step %c1  iter_args(%dma_start = %start_memcpy) -> (!equeue.signal) {
                %dma_done1 = "equeue.memcpy"(%dma_start, %act_sram, %act_mem, %dma, %k): (!equeue.signal, !equeue.container<tensor<16xf32>,i32>, !equeue.container<tensor<5xf32>, i32>, i32, index)
                %start_compute = "equeue.control_and"(%dma_done0, %dma_done1):(!equeue.signal, !equeue.signal)->!equeue.signal
                %compute_once = equeue.launch (%act_mem_local, %weight_mem_local, %ofmap_mem_local, %offset =
                %act_mem, %weight_mem, %ofmap_mem, %k: !equeue.container<tensor<5xf32>, i32>, !equeue.container<tensor<5xf32>, i32>,
                !equeue.container<f32, i32>, index)
                in (%start_compute, %aie_core_local)  {
                        %const0 = constant 0.0:f32
                        "equeue.write"(%const0, %ofmap_mem_local): (f32, !equeue.container<f32, i32>)->()
                        %cst0 = constant 0:index
                        %cst1 = constant 1:index
                        %cst5 = constant 2:index
                        scf.for %i = %cst0 to %cst5 step %cst1 {
                                %j = affine.apply affine_map<(d0,d1)->(d0+d1)>(%offset,%i)
                                %ifmap = "equeue.read" (%act_mem_local,%j):(!equeue.container<tensor<5xf32>, i32>, index)->f32
                                %filter = "equeue.read" (%weight_mem_local,%i):(!equeue.container<tensor<5xf32>, i32>, index)->f32
                                %ofmap = "equeue.read" (%ofmap_mem_local):(!equeue.container<f32, i32>) -> f32
                                %psum = mulf %filter, %ifmap: f32
                                %ofmap_flight = addf %ofmap, %psum: f32
                                "equeue.write"( %ofmap_flight, %ofmap_mem_local):(f32, !equeue.container<f32, i32>)->()
                                "scf.yield"():()->() }
                        "equeue.return"():()->() }
                %ofmap_done = "equeue.memcpy"(%compute_once, %ofmap_mem, %output_sram, %dma, %k):(!equeue.signal, !equeue.container<f32, i32>, !equeue.container<tensor<12xf32>, i32>, i32, index)
                "scf.yield"(%ofmap_done):(!equeue.signal)->() }
        "equeue.await"(%compute_done):(!equeue.signal)->()
        %act_out = "equeue.read" (%output_sram):(!equeue.container<tensor<12xf32>, i32> ) -> tensor<5xf32>
        equeue.dealloc %act_sram, %weight_sram, %output_sram, %act_mem, %weight_mem, %ofmap_mem: !equeue.container<tensor<16xf32>, i32>, !equeue.container<tensor<5xf32>, i32>, !equeue.container<tensor<12xf32
"equeue.await"(%done_signal):(!equeue.signal)->()
return %result: tensor<12xf32>
```

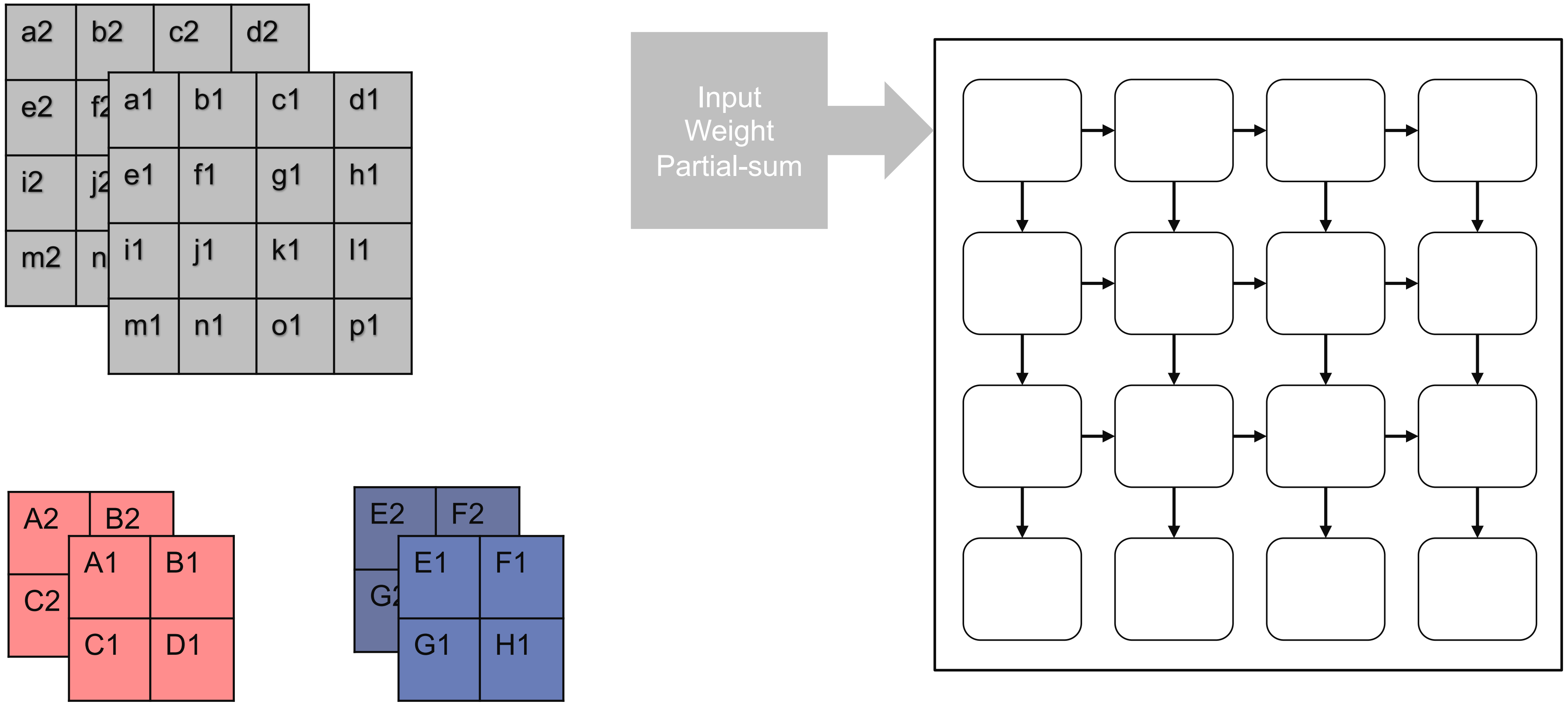
## Visualized Simulation

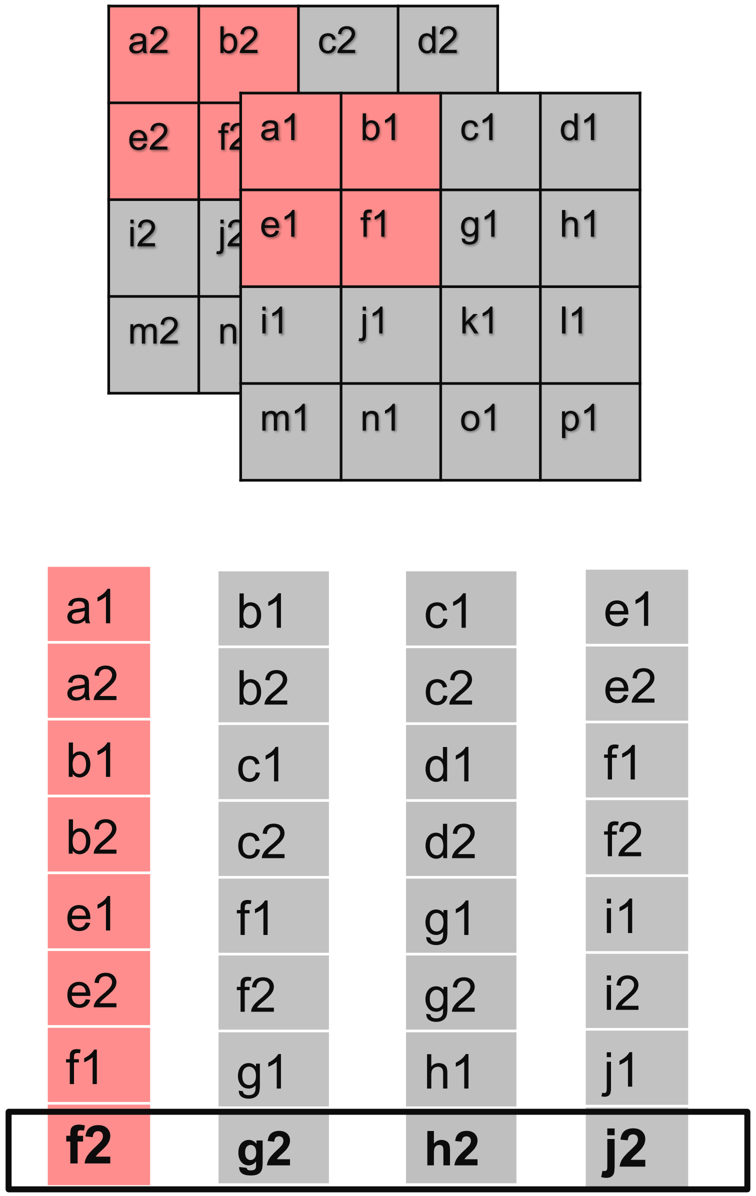


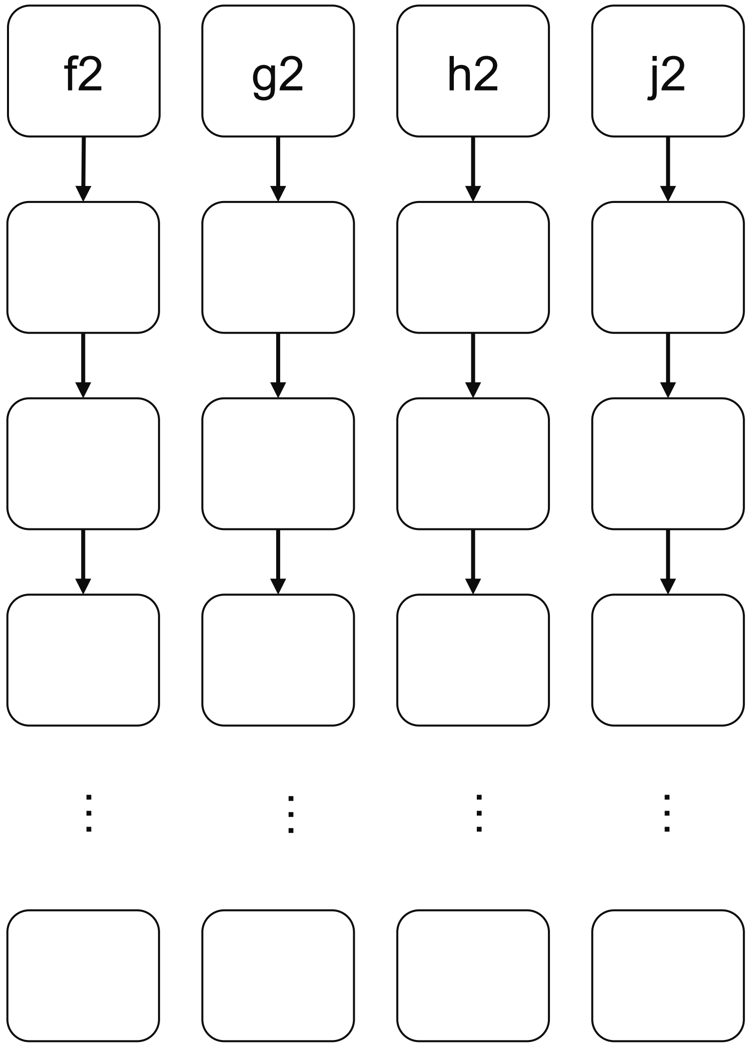
- Concurrency
- Synchronized scheduling
- Model execution time
- Await

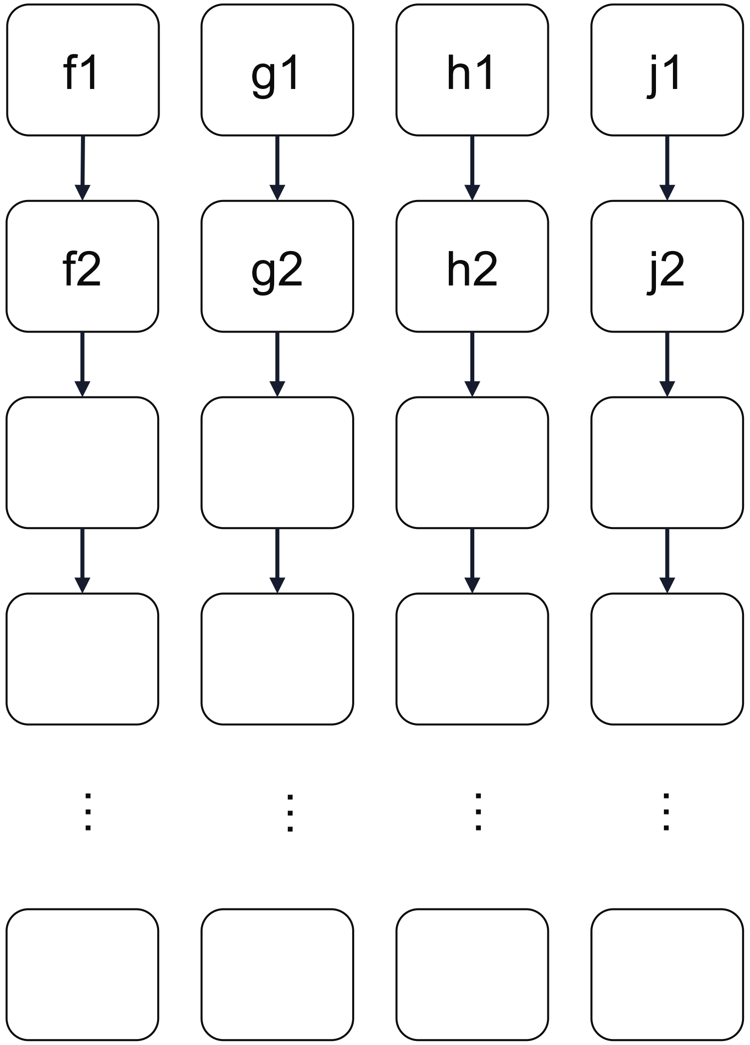
## Case Study –
## Input Stationary on Systolic Array

# Input stationary

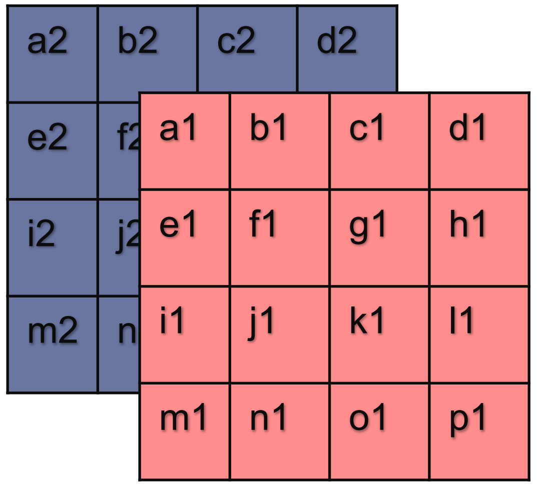


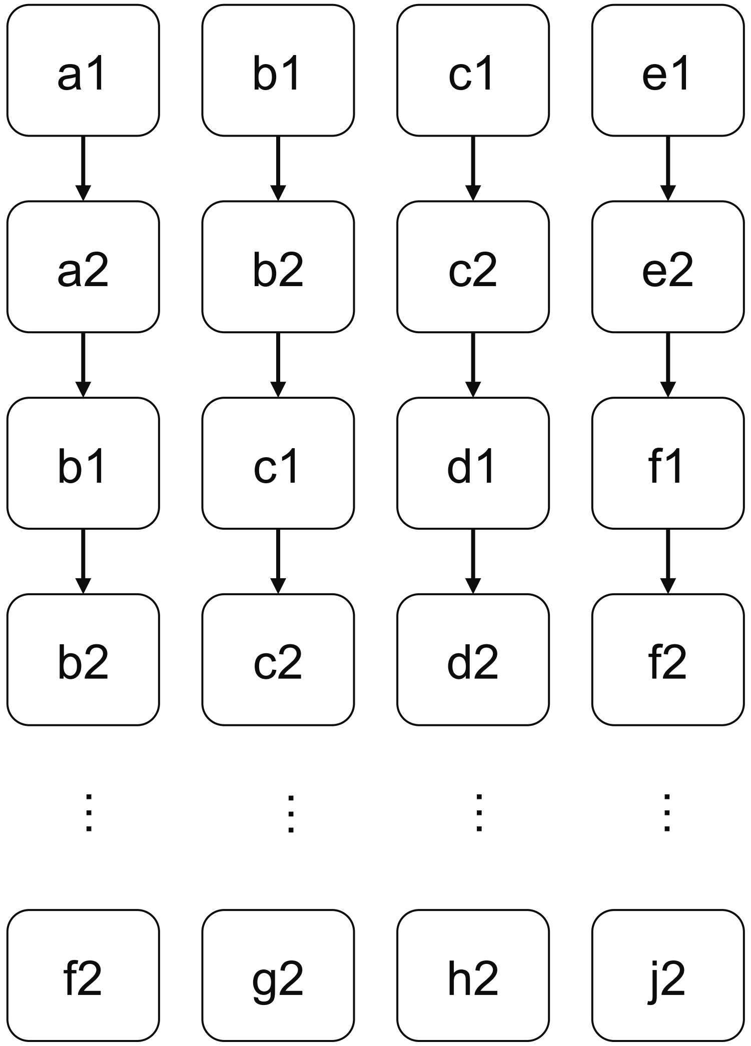

a2 b2 c2 d2
e2 f2
i2 j2
m2 n
a1 b1 c1 d1
e1 f1 g1 h1
i1 j1 k1 l1
m1 n1 o1 p1
a1 a2 b1 b2 e1 e2 f1 f2
b1 b2 c1 c2 f1 f2 g1 g2
c1 c2 d1 d2 g1 g2 h1 h2
e1 e2 f1 f2 i1 i2 j1 j2


f2
g2
h2
j2

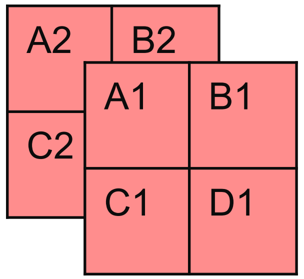

f1
g1
h1
j1
f2
g2
h2
j2

## Read inputs

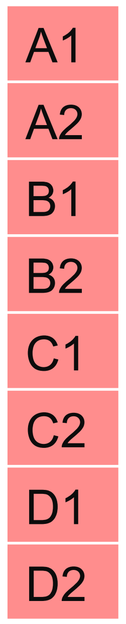

a2
b2
c2
d2
e2
f2
i2
j2
m2
n
a1
b1
c1
d1
e1
f1
g1
h1
i1
j1
k1
l1
m1
n1
o1
p1


a1
b1
c1
e1
a2
b2
c2
e2
b1
c1
d1
f1
b2
c2
d2
f2
f2
g2
h2
j2

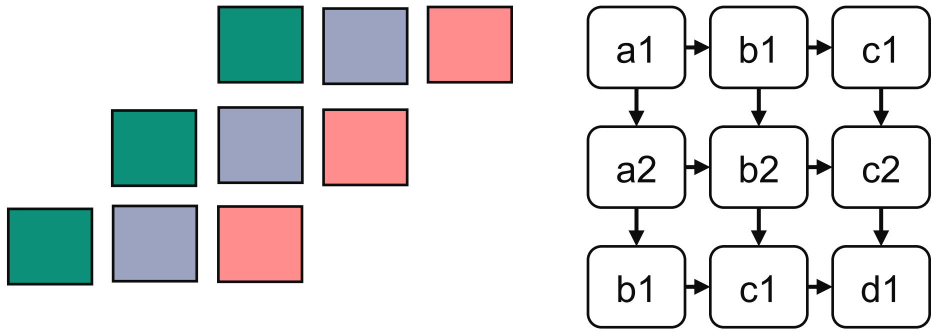
# Compute



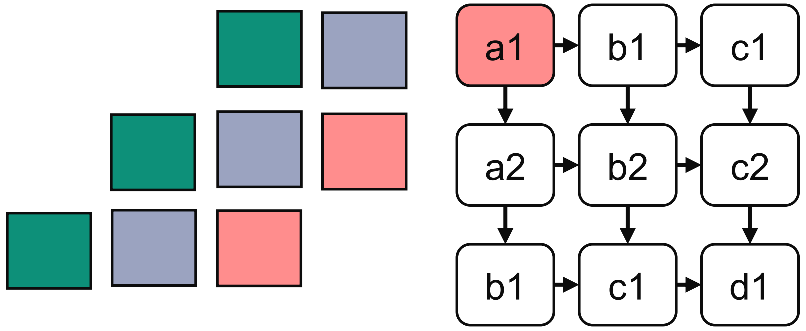
A2
B2
A1
B1
C2
C1
D1



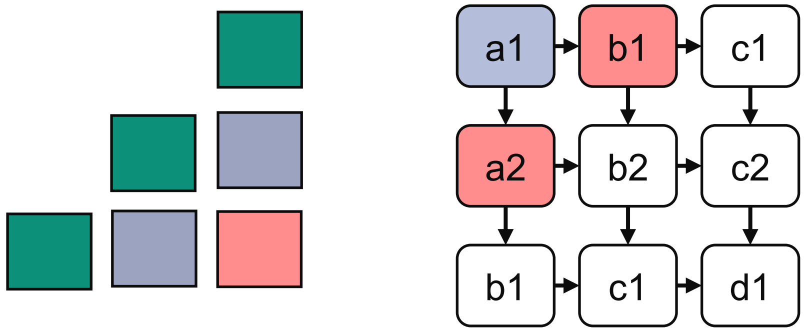
A1
A2
B1
B2
C1
C2
D1
D2



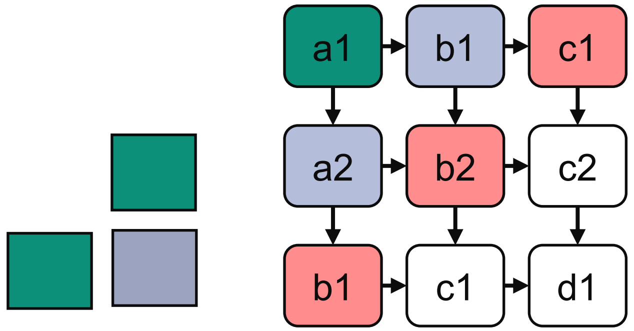
a1
b1
c1
a2
b2
c2
b1
c1
d1



a1
b1
c1
a2
b2
c2
b1
c1
d1

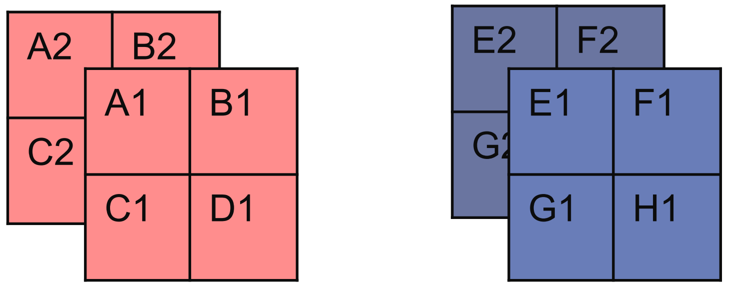


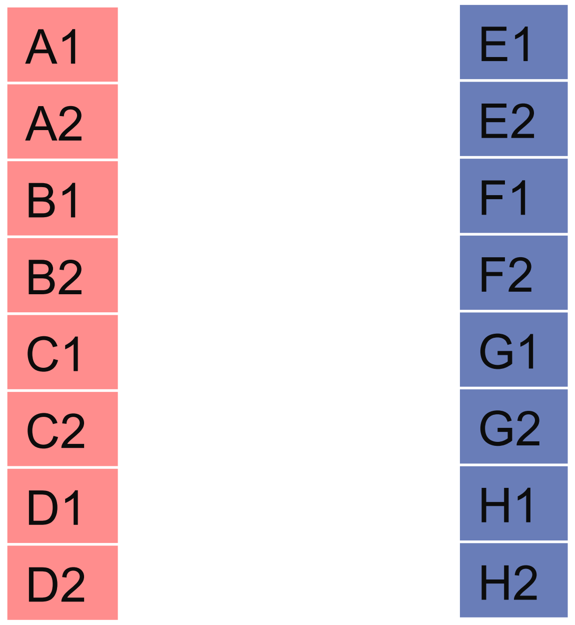
a1
b1
c1
a2
b2
c2
b1
c1
d1

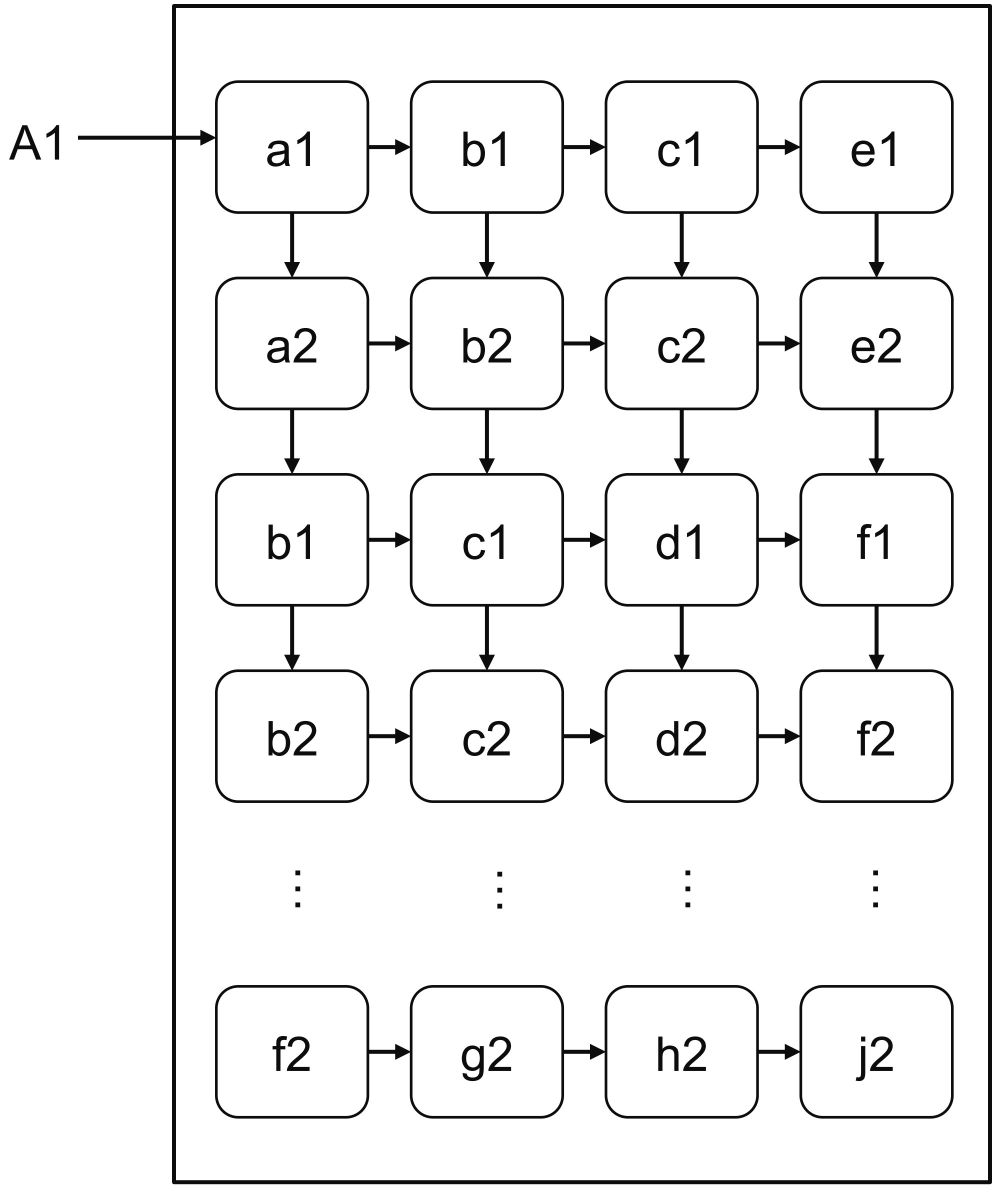


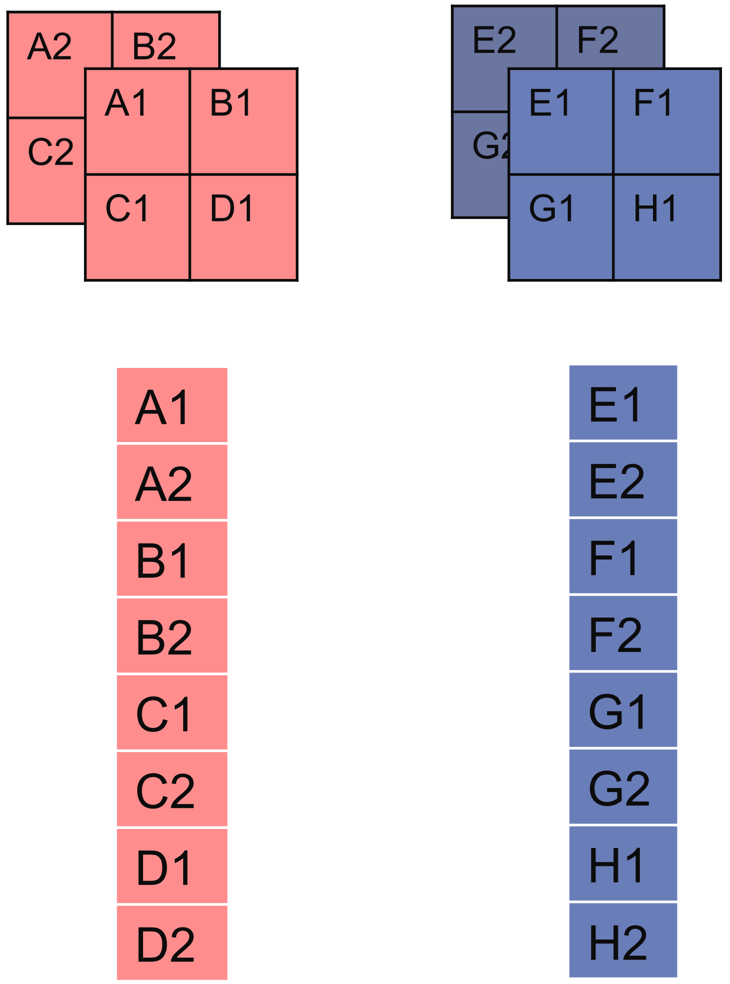
a1
b1
c1
a2
b2
c2
b1
c1
d1

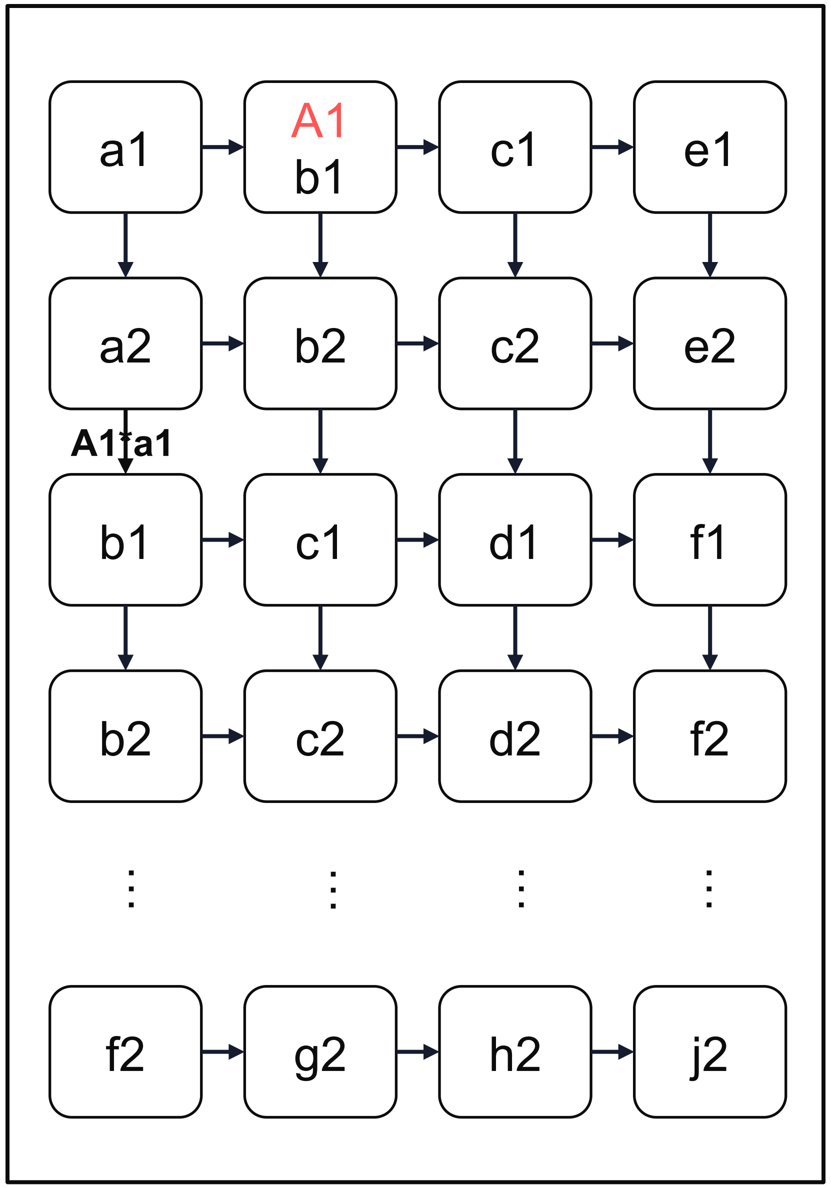
# compute

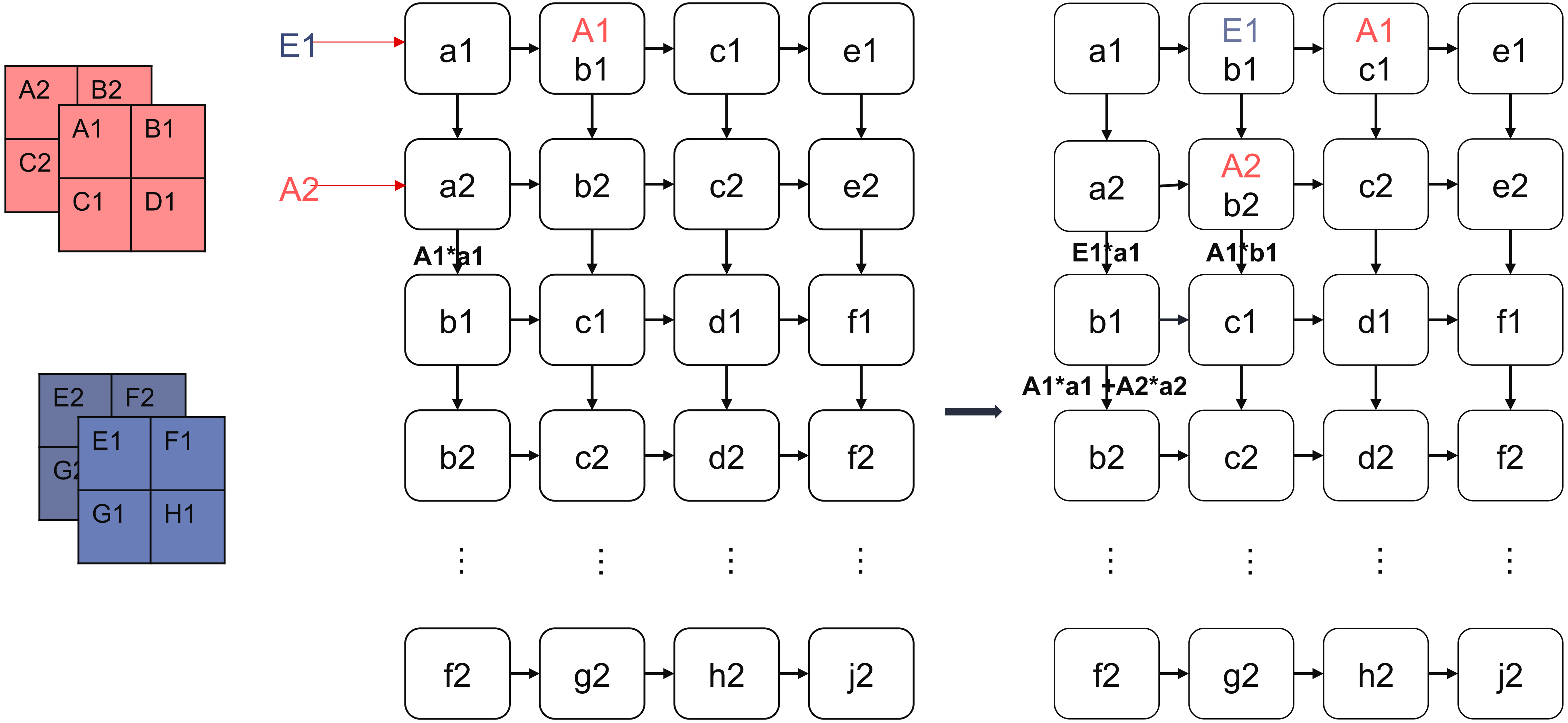

A2
B2
A1
B1
C2
C1
D1
E2
F2
E1
F1
G2
G1
H1

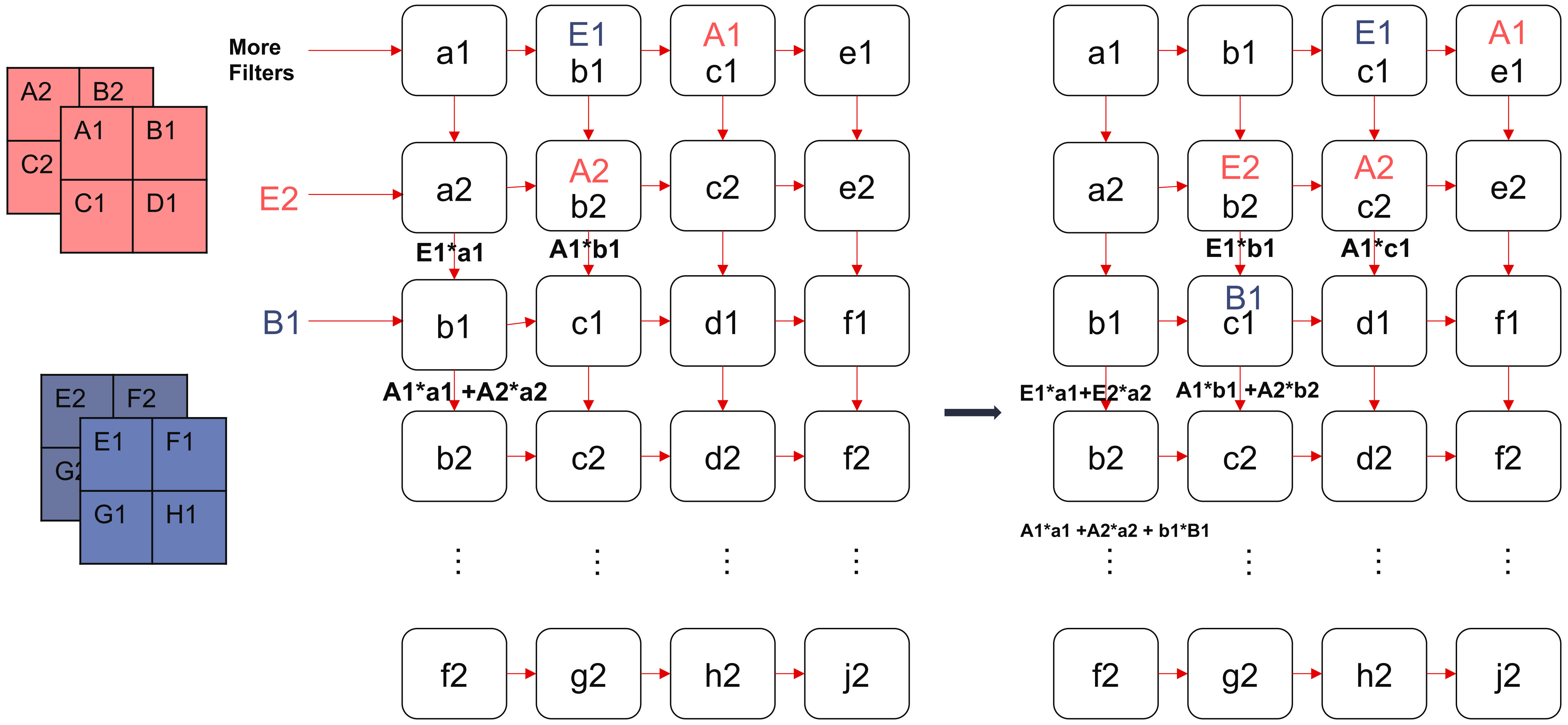

A1
A2
B1
B2
C1
C2
D1
D2
E1
E2
F1
F2
G1
G2
H1
H2


A1
a1
b1
c1
e1
a2
b2
c2
e2
b1
c1
d1
f1
b2
c2
d2
f2
f2
g2
h2
j2


A2
B2
A1
B1
C2
C1
D1
E2
F2
E1
F1
G2
G1
H1
A1
A2
B1
B2
C1
C2
D1
D2
E1
E2
F1
F2
G1
G2
H1
H2


a1
A1
b1
c1
e1
a2
b2
c2
e2
A1*a1
b1
c1
d1
f1
b2
c2
d2
f2
⋮
⋮
⋮
⋮
f2
g2
h2
j2


A2
B2
A1
B1
C2
C1
D1
E2
F2
E1
F1
G2
G1
H1
E1
A2
a1
A1
b1
c1
e1
a2
b2
c2
e2
A1*a1
b1
c1
d1
f1
b2
c2
d2
f2
f2
g2
h2
j2
a1
E1
b1
A1
c1
e1
a2
A2
b2
c2
e2
E1*a1
A1*b1
b1
c1
d1
f1
A1*a1 +A2*a2
b2
c2
d2
f2
f2
g2
h2
j2



A2
B2
A1
B1
C2
C1
D1
E2
F2
E1
F1
G2
G1
H1
More Filters
E2
B1
a1
E1 b1
A1 c1
e1
a2
A2 b2
c2
e2
E1*a1
A1*b1
b1
c1
d1
f1
A1*a1 +A2*a2
b2
c2
d2
f2
f2
g2
h2
j2
a1
b1
E1 c1
A1 e1
a2
E2 b2
A2 c2
e2
E1*b1
A1*c1
b1
B1 c1
d1
f1
E1*a1+E2*a2
A1*b1 +A2*b2
b2
c2
d2
f2
A1*a1 +A2*a2 + b1*B1
f2
g2
h2
j2

## Example Generator

- Generator create large example (mlir) with limited lines of codes (c++)
  - For loop and events
  - **declarative builders API (https://mlir.llvm.org/docs/EDSC/)**
- Generator
  - https://github.com/cucapra/event_queue/blob/scalable/lib/Generator/EQueueScaleSim.cpp
- MLIR Example
  - https://github.com/cucapra/event_queue/blob/scalable/test/EQueue/scale_sim_os.mlir
- Visualizer
  - chrome://tracing/

## Representing Parallel Flow in Equeue Dialect

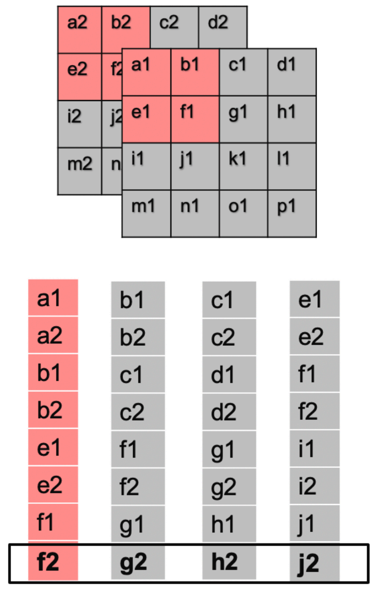
```
// start signal
start = equeue.start()

// parallel for
for h in arr_height:
        done, … =  equeue.launch(…) in (start, core[h]) {…}
        if h = 0:
                prev_done = done
        else:
                prev_done = equeue.control_and(done, prev_done)

// await for done signal to proceed
equeue.await(prev_done)
```

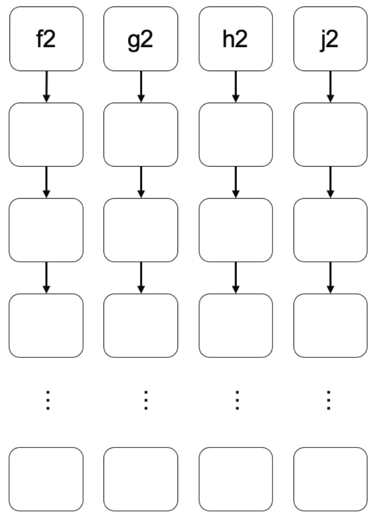
## Input Stationary Workflow

```
// Moving in inputs
for t in 0…filter_w * filter_h* channel:

    //inside each pe, read out current map

    //shift current map

    //memcopy from sram to pe registers


// compute and output results
for t in 0…output_size + arr_h + arr_w:

    // memcopy weights from sram to pe registers

    // read the current values in pe registers and calculate ofmap

    // update pe registers and potentially write output to sram
```

# Moving in inputs

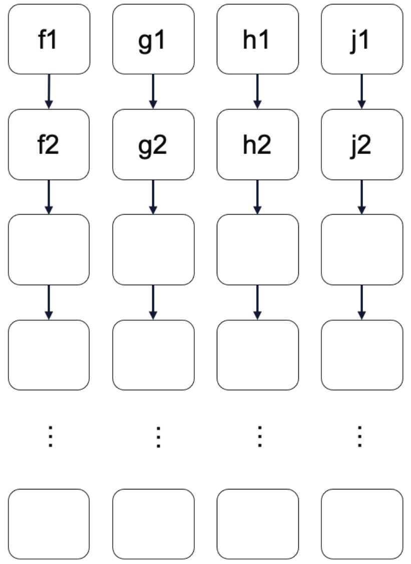
```
// Moving in inputs
for t in 0…filter_w * filter_h* channel:

    // inside each pe, read out current map
    start = equeue.start() //parallel
    for h in arr_height:
        for w in arr_width:
            done, ifmap_flight[h][w] = equeue.launch(ifmap_buffer=pe[h][w].ifmap_buffer) in (start, pe[h][w].core) {
                ifmap = equeue.read(ifmap_buffer)
                equeue.yield ifmap
            }
            if h= 0 and w = 0:
                prev_done = done
            else:
                prev_done = equeue.control_and(done, prev_done)
    equeue.await(prev_done)

    // shift current map

    // memcopy from sram to pe registers
```





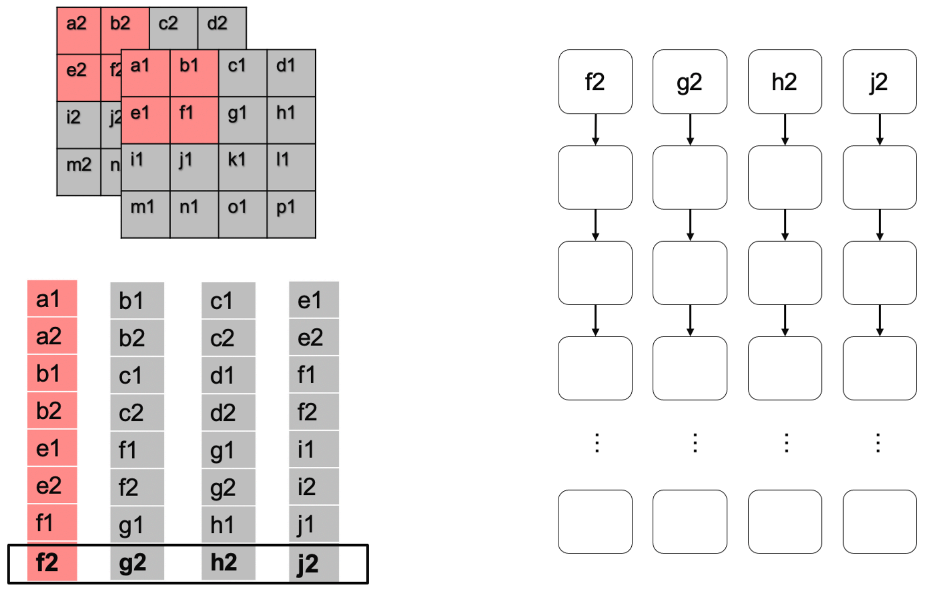
# Moving in inputs

```
// Moving in inputs
for t in 0…filter_w * filter_h* channel:

    // inside each pe, read out current map

    // shift current map
    start = equeue.start() //parallel
    for h in 1...arr_height:
        for w in arr_width:
            done = equeue.launch(ifmap_buffer=pe[h][w].ifmap_buffer, ifmap_flight=ifmap_flight[h-1][w]) in (start, pe[h][w].core) {
                equeue.write(ifmap_flight, ifmap_buffer)
            }
            if h= 1 and w = 0:
                prev_done = done
            else:
                prev_done = equeue.control_and(done, prev_done)
    equeue.await(prev_done)


    // memcopy from sram to pe registers
```



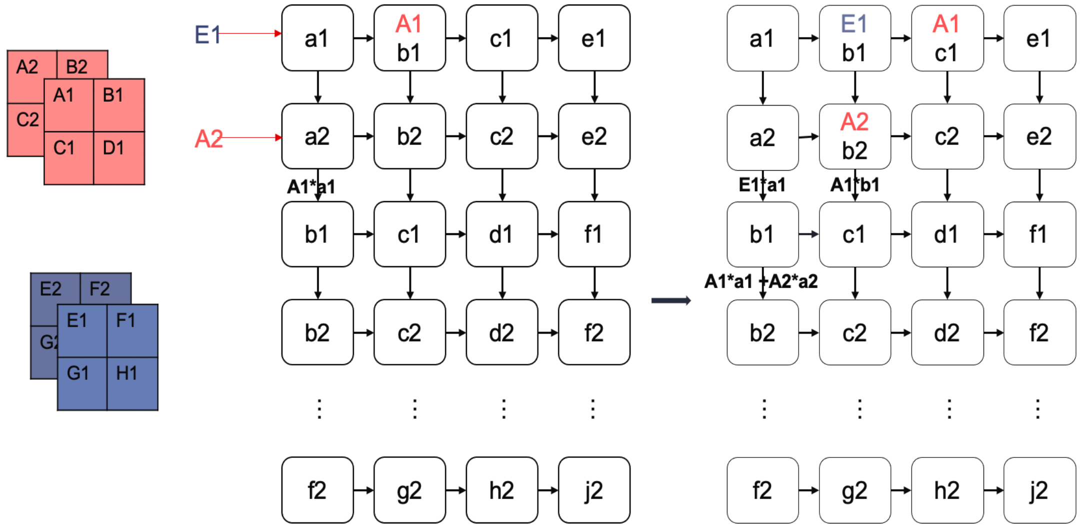
# Moving in inputs

```
// Moving in inputs
for t in 0…filter_w * filter_h* channel:

    // inside each pe, read out current map

    // shift current map

    // memcopy from sram to pe registers
    start = equeue.start()
    for w in arr_width:
        //different bank to ensure parallel execution
        done = equeue.memcopy(start, ifmap_sram, ifmap_buffer=pe[0][w].ifmap_buffer, offset, bank)
        if w = 0:
            prev_done = done
        else:
            prev_done = equeue.control_and(done, prev_done)
    equeue.await(prev_done)
```



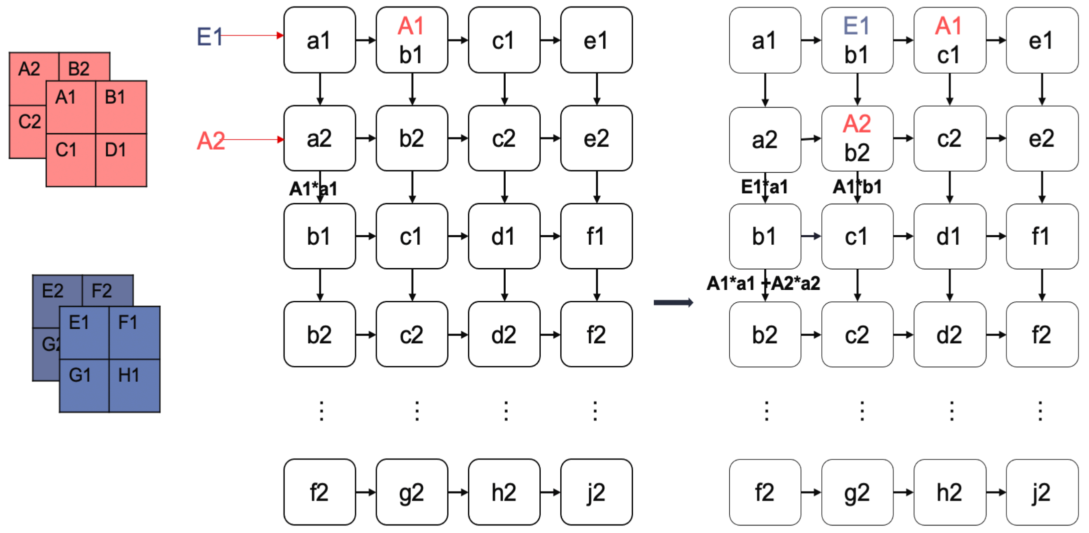
# Compute and Output Results

```
// compute and output results
for t in 0...output_size + arr_h + arr_w:

    // memcopy weights from sram to pe registers
    start = equeue.start()
    for n in 0...weights_num (step=write_direction):
        done = equeue.memcopy(start, weights_sram, weight_buffer=pe[n][0].weight_buffer, offset, bank)
        if w = 0:
            prev_done = done
        else:
            prev_done = equeue.control_and(done, prev_done)
    equeue.await(prev_done)

    // read the current values in pe registers and calculate ofmap

    // update pe registers and potentially write output to sram
```



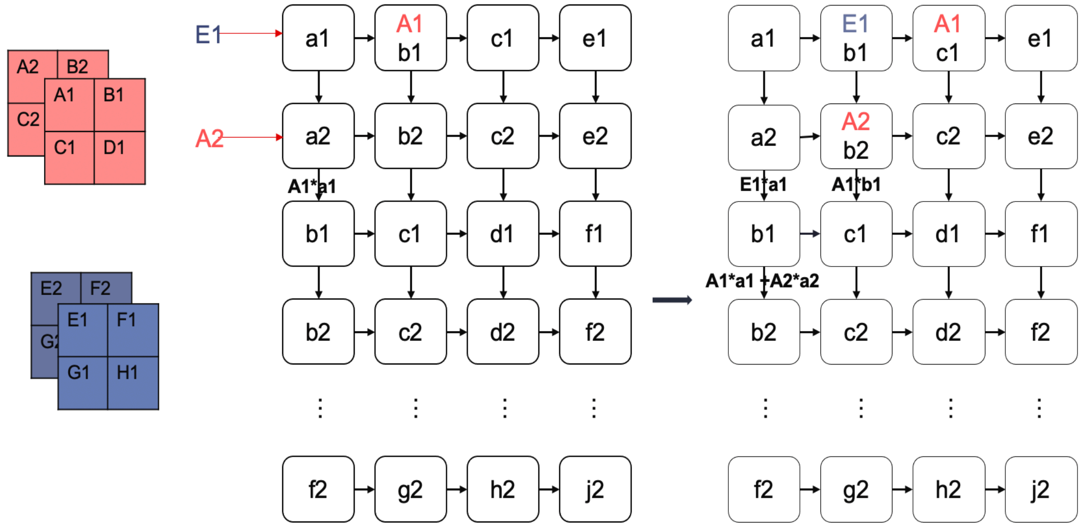
# Compute and Output Results

```
// compute and output results
for t in 0…output_size + arr_h + arr_w:

    // memcopy weights from sram to pe registers

    // read the current values in pe registers and calculate ofmap
    start = equeue.start()
    for h in arr_height:
        for w in arr_width:
            done, weight_flight[h][w], ofmap_flight[h][w]= equeue.launch( start, ifmap_buffer=pe[h][w].ifmap_buffer,
                ofmap_buffer=pe[h][w].ofmap_buffer,
                weight_buffer=pe[h][w].weight_buffer
                ) in (start, pe[h][w].core) {
                ifmap = ifmap_buffer
                weight = weight_buffer
                ofmap_old = equeue.read(ofmap_buffer)
                ofmap = ifmap * weight + ofmap_old
                equeue.yield weights, ofmap
            }
            if h= 0 and w = 0: prev_done = done
            else:  prev_done = equeue.control_and(done, prev_done)
    equeue.await(prev_done)

    // update pe registers and potentially write output to sram
```



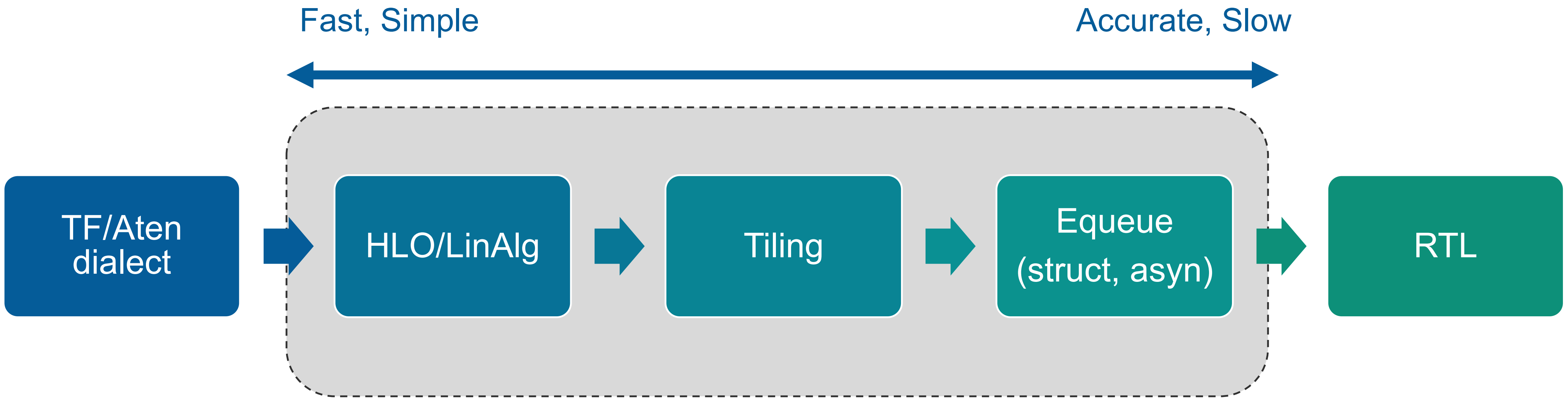
# Compute and Output Results

```
// compute and output results
for t in 0…output_size + arr_h + arr_w:

  // memcopy weights from sram to pe registers

  // read the current values in pe registers and calculate ofmap

  // update pe registers and potentially write output to sram
  start = equeue.start()
  for r in arr_height:
    for c in arr_width:
      done = equeue.launch(
        start,
        wflight=wfmap_flight[r][c],
        oflight=ofmap_flight[r][c],
        wbuffer=wbuffer2s[r][c+1],
        obuffer=obuffer2s[r+1][c],
        ){
        //no outputs
        equeue.write(oflight, obuffer)
        equeue.write(wflight, wbuffer)
        if r+c <= t and t <= ofmap_size + arr_h +c:
          equeue.writeop(oflight, out_sram)
        }
    }
  equeue.await(prev_done)
```



## High Level Abstractions

## Lowering Pipeline

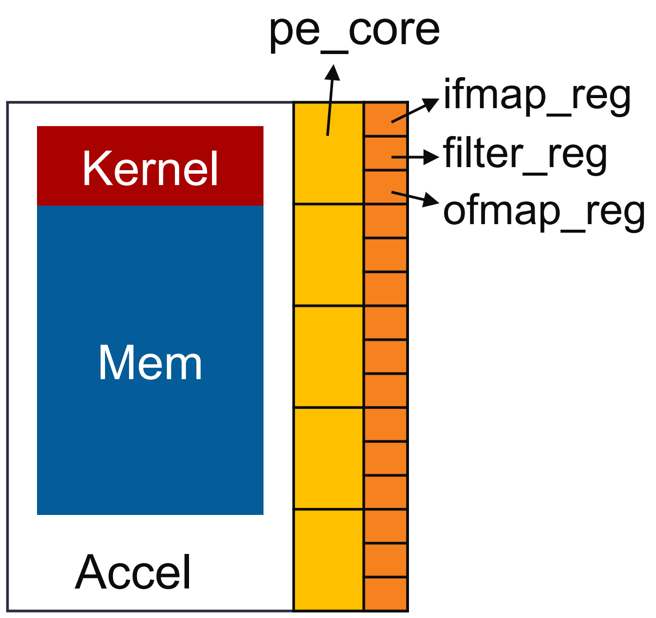


## Different Level of Abstractions in EQueue Dialect

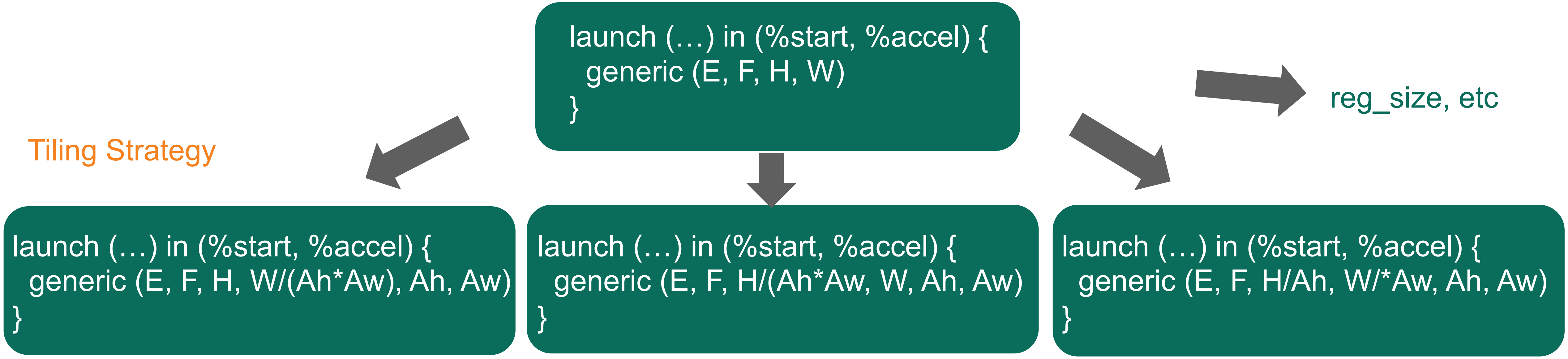
```
launch (…) in (%start, %accel) {
  generic (E, F, H, W)
}
```

## Different Level of Abstractions in EQueue Dialect

```
launch (…) in (%start, %accel) {
  generic (E, F, H, W)
}
```



# Different Level of Abstractions in EQueue Dialect



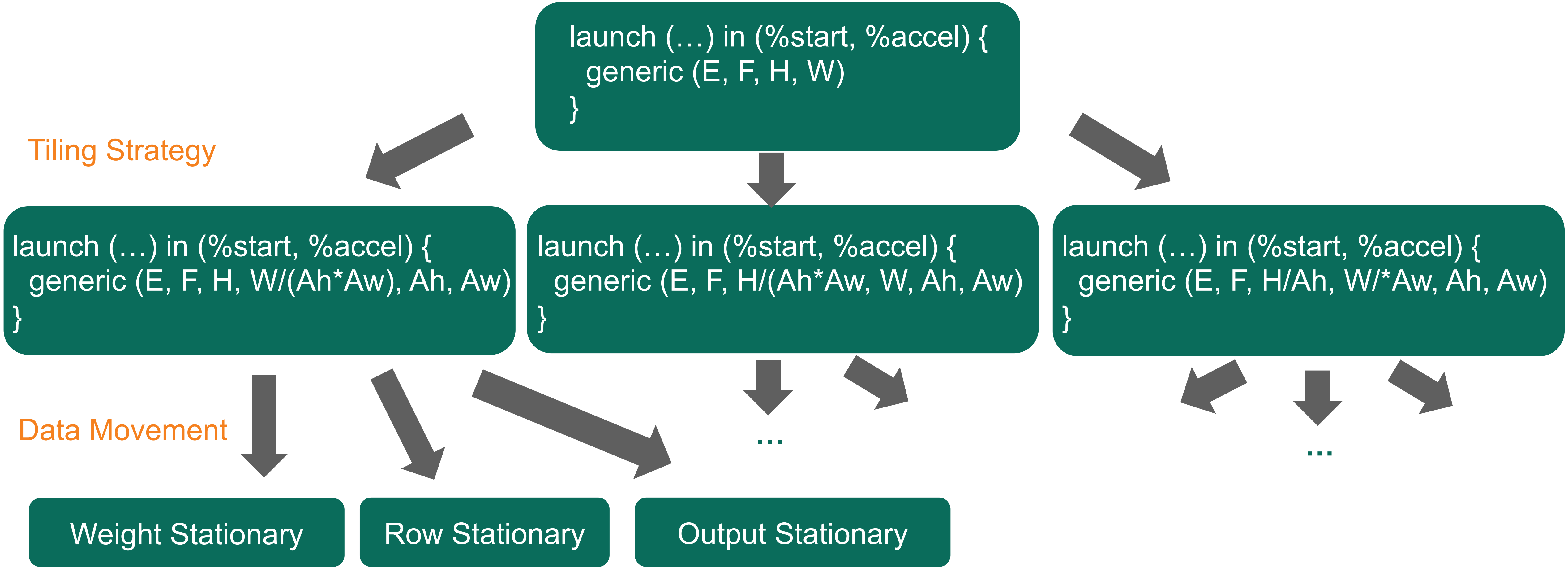
```
launch (…) in (%start, %accel) {
  generic (E, F, H, W)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H, W/(Ah*Aw), Ah, Aw)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H/(Ah*Aw, W, Ah, Aw)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H/Ah, W/*Aw, Ah, Aw)
}
```

# Different Level of Abstractions in EQueue Dialect



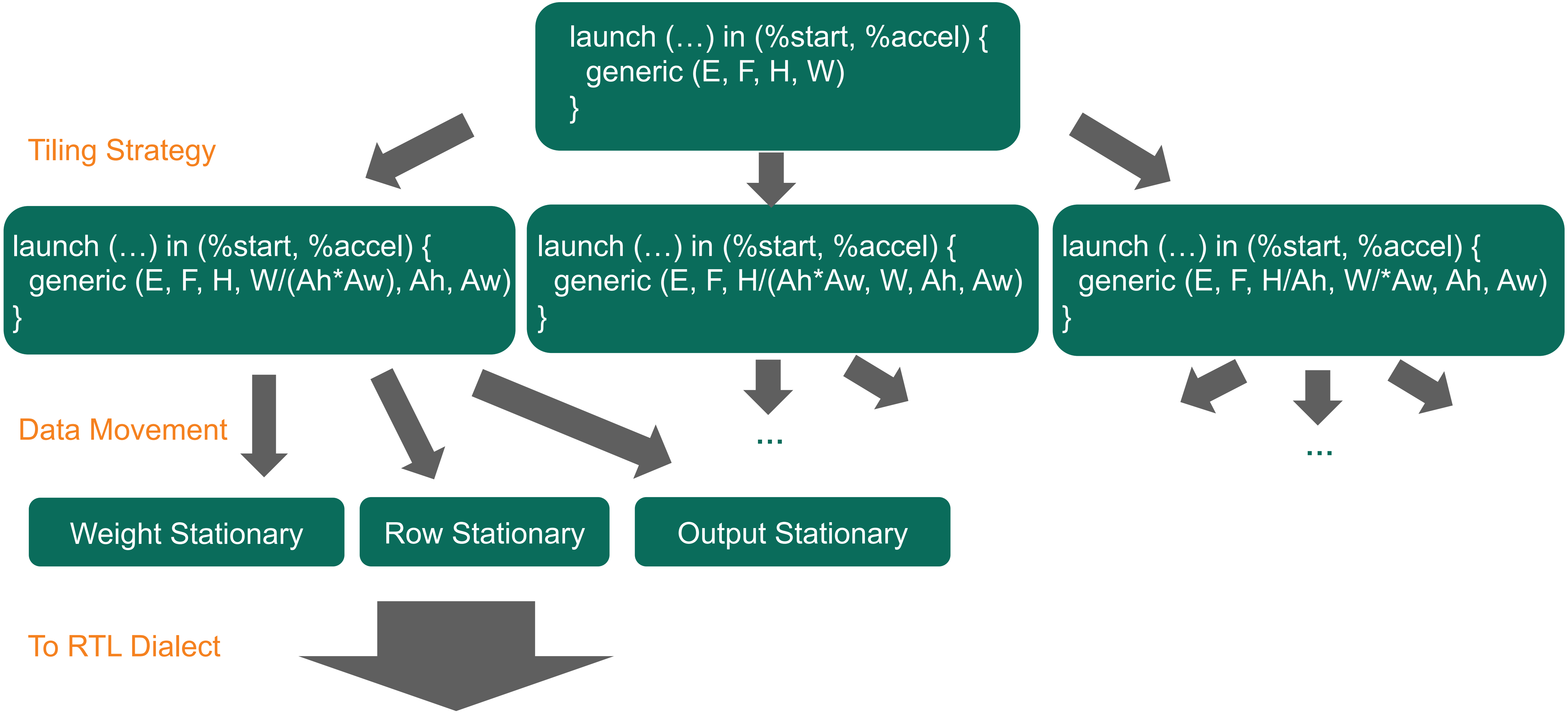
```
launch (…) in (%start, %accel) {
  generic (E, F, H, W)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H, W/(Ah*Aw), Ah, Aw)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H/(Ah*Aw, W, Ah, Aw)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H/Ah, W/*Aw, Ah, Aw)
}
```

# Different Level of Abstractions in EQueue Dialect



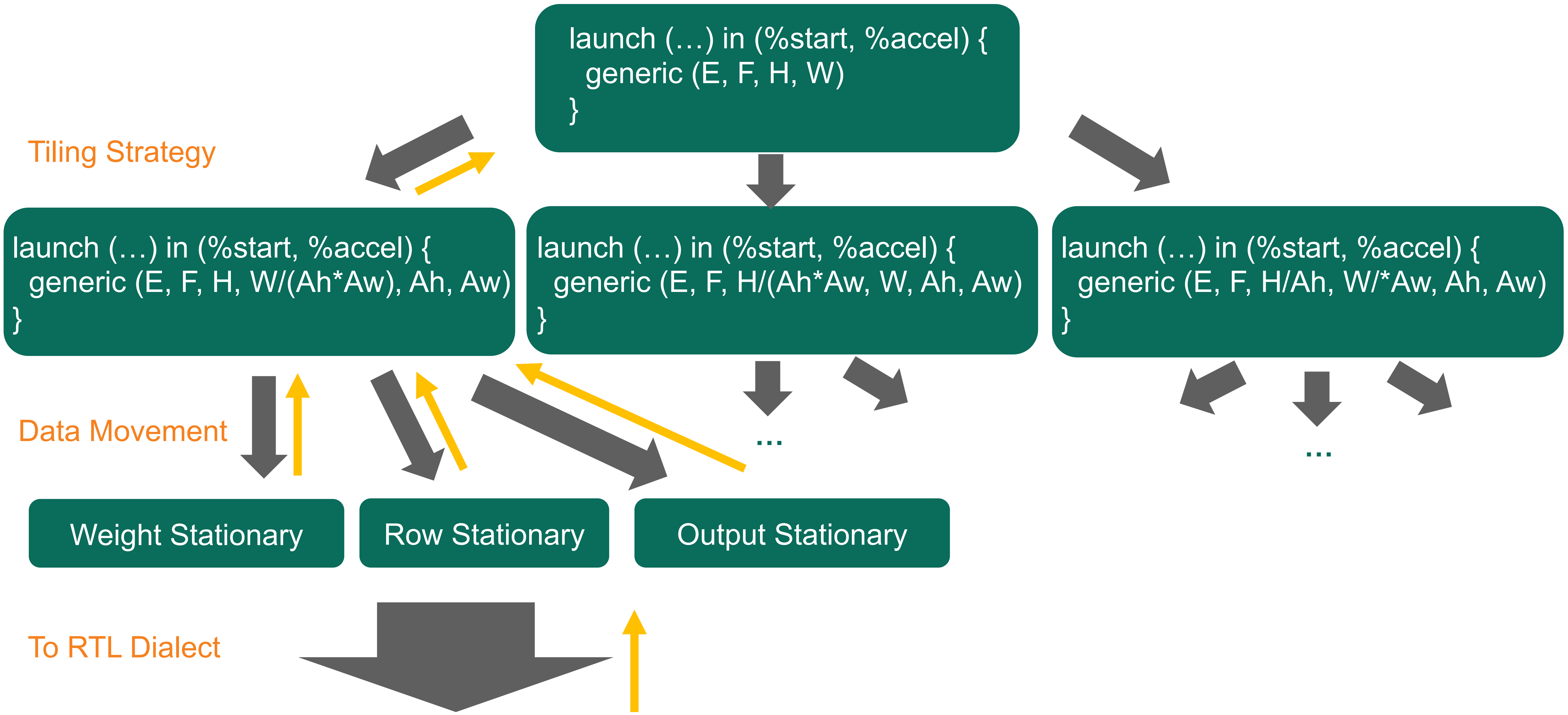
```
launch (…) in (%start, %accel) {
  generic (E, F, H, W)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H, W/(Ah*Aw), Ah, Aw)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H/(Ah*Aw, W, Ah, Aw)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H/Ah, W/*Aw, Ah, Aw)
}
```

# Different Levels of Abstractions in EQueue Dialect



```
launch (…) in (%start, %accel) {
  generic (E, F, H, W)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H, W/(Ah*Aw), Ah, Aw)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H/(Ah*Aw, W, Ah, Aw)
}
```

```
launch (…) in (%start, %accel) {
  generic (E, F, H/Ah, W/*Aw, Ah, Aw)
}
```

Weight Stationary

Row Stationary

Output Stationary

## Background: Linalg Dialect

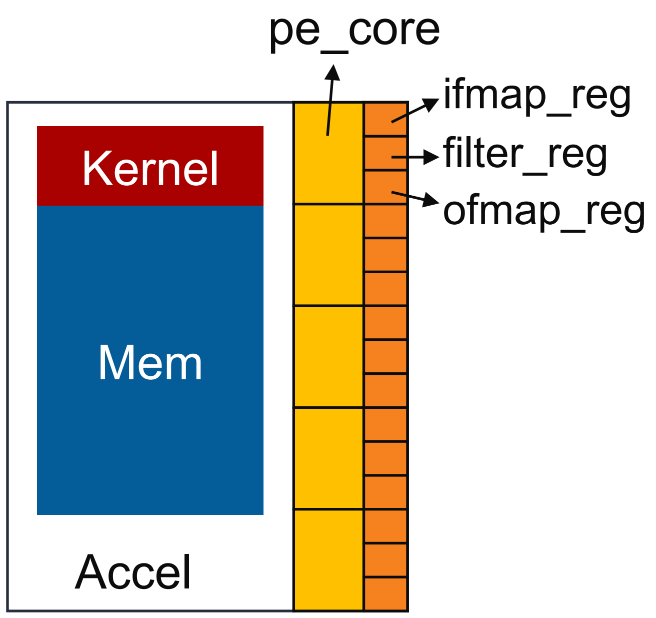
```
#accesses = [
  affine_map<(E, F, H, W) -> (E, F)>,
  affine_map<(E, F, H, W) -> (H, W)>,
  affine_map<(E, F, H, W) -> (E, F)>
]
#trait = {
  args_in = 2,
  args_out = 1,
  iterator_types = ["reduction", "reduction", "parallel", "parallel"],
  indexing_maps = #accesses
}
func @compute(%A: memref<3x3xf32>,%B: memref<5x5xf32>, %C:
memref<3x3xf32>) {
  linalg.generic #trait %A, %B, %C {
    ^bb0(%a: f32, %b: f32, %c: f32):
      %d = "some_compute"(%a, %b): (f32, f32)->f32
      linalg.yield %d: f32
  } : memref<3x3xf32>, memref<5x5xf32>, memref<3x3xf32>
  return
}
```

## Background: Representation of Linalg in Loops

```
func @compute(%arg0: memref<3x3xf32>, %arg1: memref<5x5xf32>, %arg2: memref<3x3xf32>) {
  %c3 = constant 3 : index
  %c5 = constant 5 : index
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  scf.for %arg3 = %c0 to %c3 step %c1 {
    scf.for %arg4 = %c0 to %c3 step %c1 {
      scf.parallel (%arg5, %arg6) = (%c0, %c0) to (%c5, %c5) step %c1 {
        %0 = load %arg0[%arg3, %arg4] : memref<3x3xf32>
        %1 = load %arg1[%arg5, %arg6] : memref<5x5xf32>
        %2 = "some_compute"(%0, %1) : (f32, f32) -> f32
        store %2, %arg2[%arg3, %arg4] : memref<3x3xf32>
        scf.yield
      }
    }
  }
  return
}
```

## Equeue - Highest Level

```
#accesses = [
  affine_map<(E, F, H, W) -> (E, F)>,
  affine_map<(E, F, H, W) -> (H, W)>,
  affine_map<(E, F, H, W) -> (E, F)>
]
#trait = {
  args_in = 2,
  args_out = 1,
  iterator_types = ["reduction", "reduction", "parallel", "parallel"],
  indexing_maps = #accesses
}
equeue.launch(%A=%ifmap, %B = %filter, %C = %ofmap) in (%start_signal, %accel_kernel){
  linalg.generic #trait %A, %B, %C {
    ^bb0(%a: f32, %b: f32, %c: f32):
      %d = "some_compute"(%a, %b): (f32, f32)->f32
      linalg.yield %d: f32
  } : memref<3x3xf32>, memref<5x5xf32>, memref<3x3xf32>
  equeue.return
}
```



# Tiling Linalg

```
#map0 = affine_map<(d0) -> (1, -d0 + 3)>
#map1 = affine_map<(d0, d1)[s0] -> (d0 * 3 + s0 + d1)>
#map2 = affine_map<(d0) -> (1, -d0 + 5)>
#map3 = affine_map<(d0, d1)[s0] -> (d0 * 5 + s0 + d1)>
#map4 = affine_map<(d0, d1, d2, d3) -> (d0, d1)>
#map5 = affine_map<(d0, d1, d2, d3) -> (d2, d3)>
```

```
equeue.launch(%arg0=%ifmap, %arg1 = %filter,
%arg2 = %ofmap) in (%start_signal, %accel_kernel){
  … // constants
  scf.for %arg3 = %c0 to %c3 step %c1 {
      scf.for %arg4 = %c0 to %c3 step %c1 {
        scf.parallel (%arg5) = (%c0) to (%c5) step (%c1) {
          %0 = affine.min #map0(%arg3)
          %1 = affine.min #map0(%arg4)
          %2 = subview %arg0[%arg3, %arg4] [%0, %1] [1, 1]  : memref<3x3xf32> to memref<?x?xf32, #map1>
          %3 = affine.min #map2(%arg5)
          %4 = subview %arg1[0, %arg5] [5, %3] [1, 1]  : memref<5x5xf32> to memref<5x?xf32, #map3>
          %5 = affine.min #map0(%arg3)
          %6 = affine.min #map0(%arg4)
          %7 = subview %arg2[%arg3, %arg4] [%5, %6] [1, 1]  : memref<3x3xf32> to memref<?x?xf32, #map1>
          linalg.generic {args_in = 2 : i64, args_out = 1 : i64, indexing_maps = [#map4, #map5, #map4], iterator_types =
["reduction", "reduction", "parallel", "parallel"]} %2, %4, %7 {
          ^bb0(%arg6: f32, %arg7: f32, %arg8: f32):  // no predecessors
            %8 = "some_compute"(%arg6, %arg7) : (f32, f32) -> f32
            linalg.yield %8 : f32
          }: memref<?x?xf32, #map1>, memref<5x?xf32, #map3>, memref<?x?xf32, #map1> }}}
    equeue.return }
```

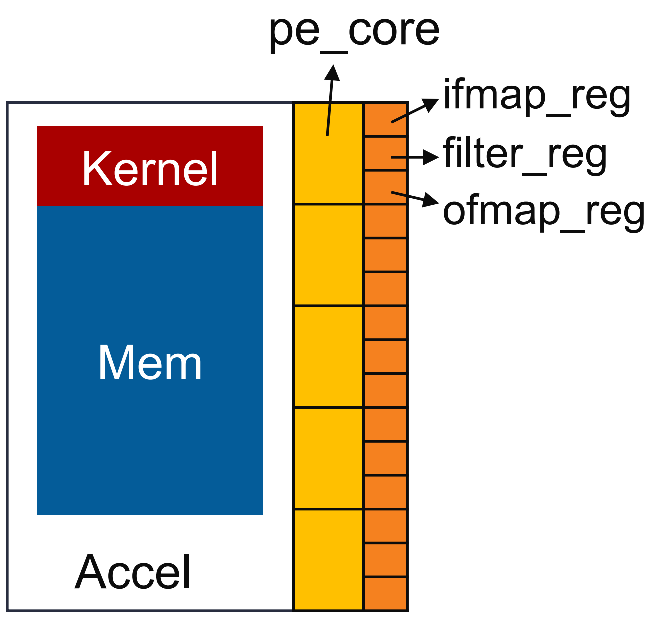
## Tiling Linalg

```
equeue.launch(%arg0=%ifmap, %arg1 = %filter,
%arg2 = %ofmap) in (%start_signal, %accel_kernel){
  … // constants
  scf.for %arg3 = %c0 to %c3 step %c1 {
     scf.for %arg4 = %c0 to %c3 step %c1 {
       scf.parallel (%arg5) = (%c0) to (%c5) step (%c1) {

          …

       }: memref<?x?xf32, #map1>, memref<5x?xf32, #map3>, memref<?x?xf32, #map1> }}}
    equeue.return }
```

```
#map0 = affine_map<(d0) -> (1, -d0 + 3)>
#map1 = affine_map<(d0, d1)[s0] -> (d0 * 3 + s0 + d1)>
#map2 = affine_map<(d0) -> (1, -d0 + 5)>
#map3 = affine_map<(d0, d1)[s0] -> (d0 * 5 + s0 + d1)>
#map4 = affine_map<(d0, d1, d2, d3) -> (d0, d1)>
#map5 = affine_map<(d0, d1, d2, d3) -> (d2, d3)>
```

# Parallel Loop Requires Control-Structure Mapping!

```
equeue.launch(%arg0=%ifmap, %arg1 = %filter, %arg2 = %ofmap) in (%start_signal,
%accel_kernel){
  … // constants
  scf.for %arg3 = %c0 to %c3 step %c1 {
      scf.for %arg4 = %c0 to %c3 step %c1 {
        %start_pe = equeue.control_start()
        equeue.launch(%local_ifmap=%pe0_ifmap, %local_filter=%pe0_filter,
%local_ofmap=%pe0_ofmap) in (%start_pe, %pe0_core){
          linalg.generic {args_in = 2 : i64, args_out = 1 : i64, indexing_maps = [#map4,
#map5, #map4], iterator_types = ["reduction", "reduction", "parallel", "parallel"]}
%local_ifmap, %local_filter, %local_ofmap {
          ^bb0(%arg6: f32, %arg7: f32, %arg8: f32):  // no predecessors
            %8 = "some_compute"(%arg6, %arg7) : (f32, f32) -> f32
            linalg.yield %8 : f32
          }: memref<?x?xf32, #map1>, memref<5x?xf32, #map3>, memref<?x?xf32,
#map1> }

        equeue.launch(%local_ifmap=%pe1_ifmap, %local_filter=%pe1_filter,
%local_ofmap=%pe1_ofmap) in (%start_pe, %pe1_core){…}

        … //pe2, pe3, pe4
  }}
equeue.return }
```



# Parallel Loops to Equeue Structure – Data Movements

```
equeue.launch(%arg0=%ifmap, %arg1 = %filter, %arg2 = %ofmap) in (%start_signal, %accel_kernel){
  … // constants
scf.for %arg3 = %c0 to %c3 step %c1 {
      scf.for %arg4 = %c0 to %c3 step %c1 {
        %start_pe = equeue.control_start()
        %done_copy0 = equeue.memcpy(%start_pe, %2, %local_ifmap,
                                    %offset0, %bank0)
        %done_copy1 = equeue.memcpy(%start_pe, %2, %local_ifmap,
                                    %offset0, %bank0)
        …

        equeue.launch(%local_ifmap=%pe0_ifmap, %local_filter=%pe0_filter,
        %local_ofmap=%pe0_ofmap) in (%done_copy0, %pe0_core){…}
        equeue.launch(%local_ifmap=%pe1_ifmap, %local_filter=%pe1_filter,
        %local_ofmap=%pe1_ofmap) in (%done_copy1, %pe1_core){…}
         … //pe1, .. pe2, pe3, pe4
  }}
equeue.return }
```

## Parallel Loops to Equeue Structure – Weight Stationary

```
equeue.launch(%arg0=%ifmap, %arg1 = %filter, %arg2 = %ofmap) in (%start_signal, %accel_kernel){
  … // constants
  scf.for %arg3 = %c0 to %c3 step %c1 {
    scf.for %arg4 = %c0 to %c3 step %c1 {
      %start_pe = equeue.control_start()
      %done_copy0 = equeue.memcpy(%start_pe, %2, %local_ifmap,
                                      %offset0, %bank0)
      %done_copy1 = equeue.memcpy(%start_pe, %2, %local_ifmap,
                                      %offset0, %bank0)
      …
      %done_copy = equeue.await(%done_copy0, %done_copy1, …)
      scf.for %arg4 = %c0 to %c5 step %c1 {
              equeue.launch(%local_ifmap=%pe0_ifmap, %local_filter=%pe0_filter,
              %local_ofmap=%pe0_ofmap) in (%done_copy, %pe0_core){…}
              equeue.launch(%local_ifmap=%pe1_ifmap, %local_filter=%pe1_filter,
              %local_ofmap=%pe1_ofmap) in (%done_copy, %pe1_core){…}
               … //pe1, .. pe2, pe3, pe4
      }
  }}
equeue.return }
```

## Ongoing Work & Future Directions

- Different levels of abstractions
- Generate fast simulation
- Functional correctness verification
- Representing dynamic execution
  - Sparsity!

Thank you!