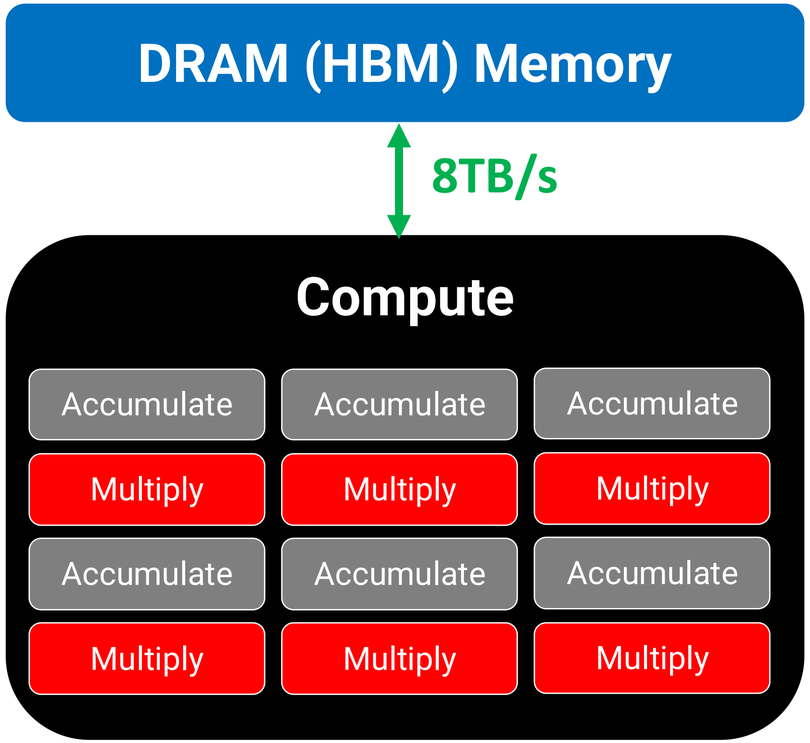
MLIR Graph Compiler for In-Memory Inference Computing

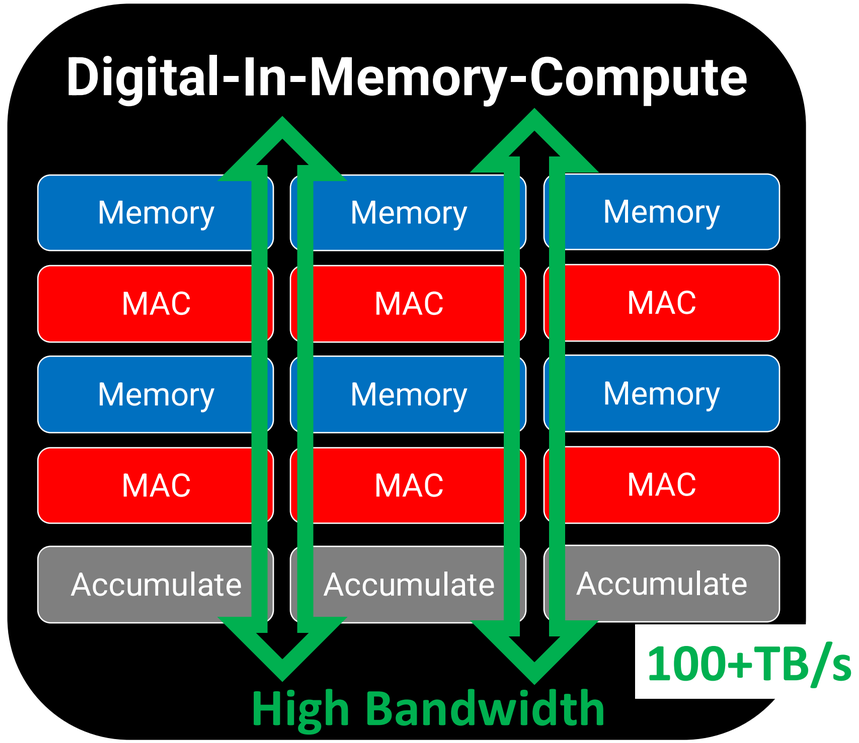
Satyam Srivastava, Kshitij Jain, Vinay M, Prashantha NR, Sudeep Bhoja
[d-Matrix Corporation]

A New Computing Paradigm for GenAI Inference



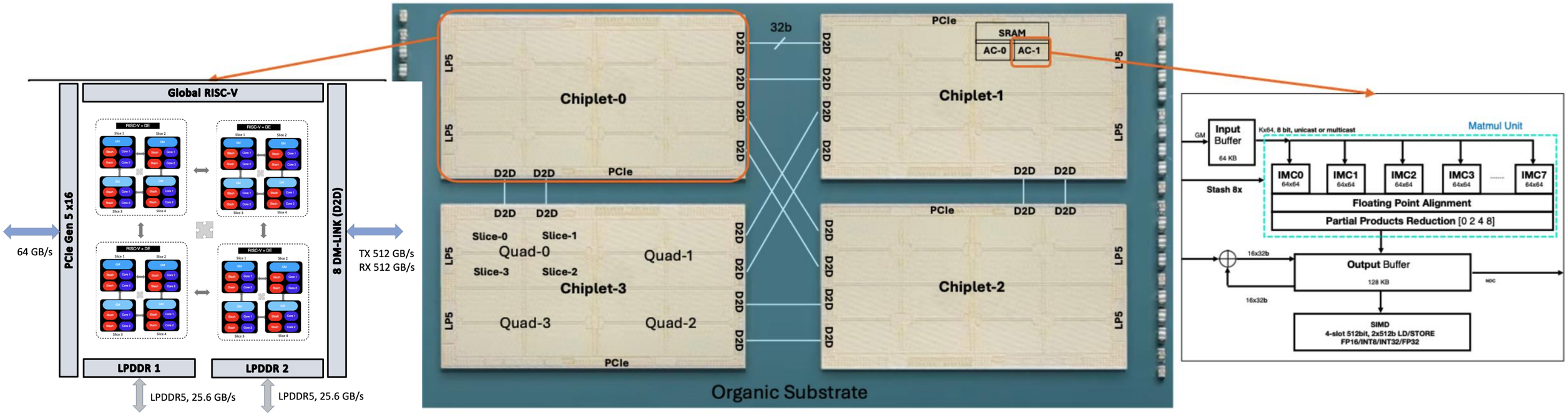
Traditional architecture

The A.I. Barrier



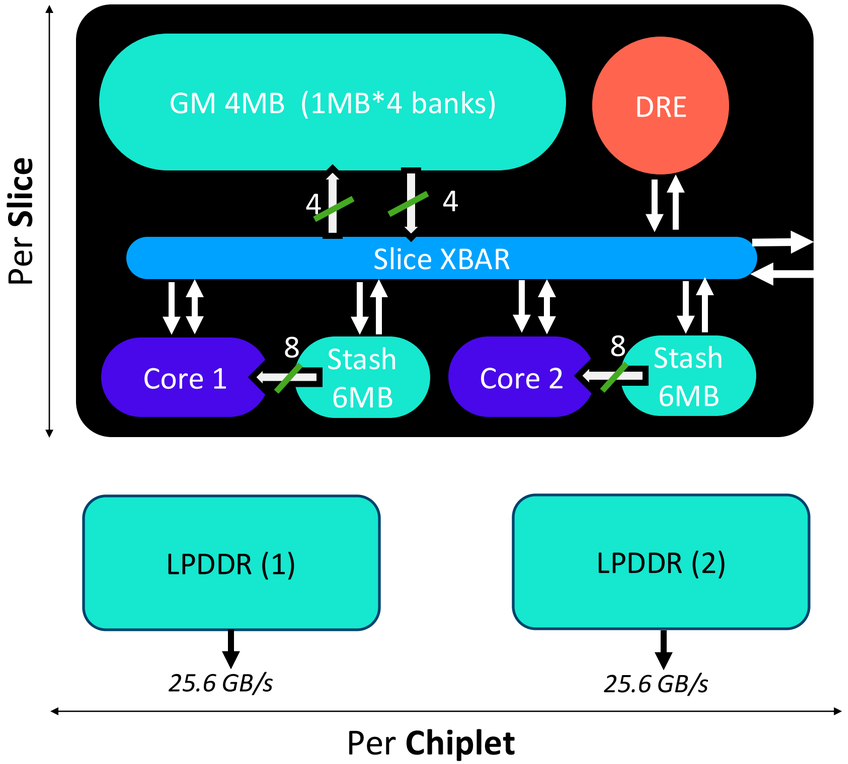
d-Matrix architecture

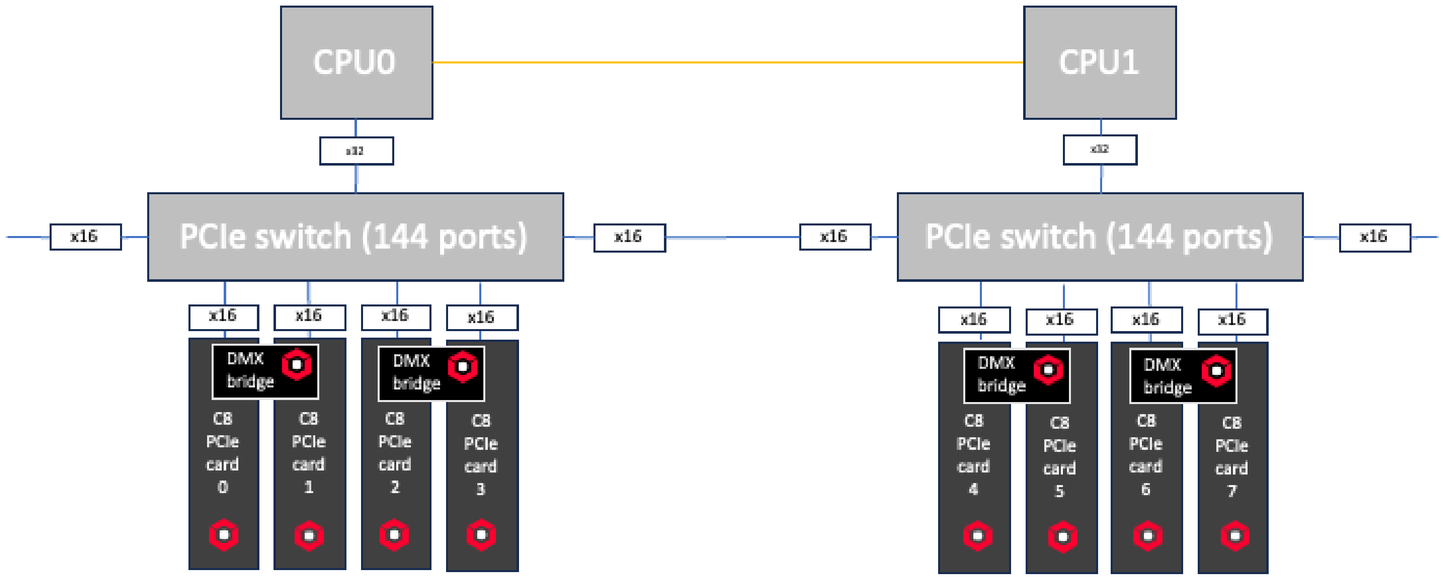
Corsair Accelerator Architecture Purpose Built for AI



Chiplet-based modular design: connectivity, scaling, cost
In-memory compute with block floating point: efficiency, throughput, accuracy
Large, on-chip performance memory: high bandwidth for params and caches
Independent, collaborative Quads: asynchronous, self-contained graph execution

Harnessing Memory Hierarchy and Scalability

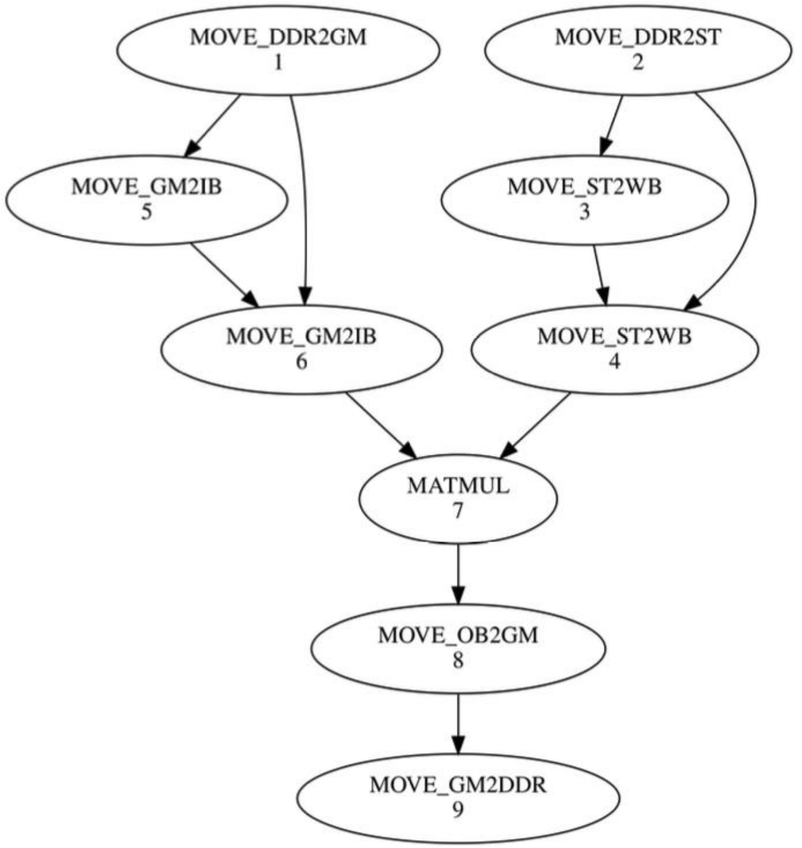




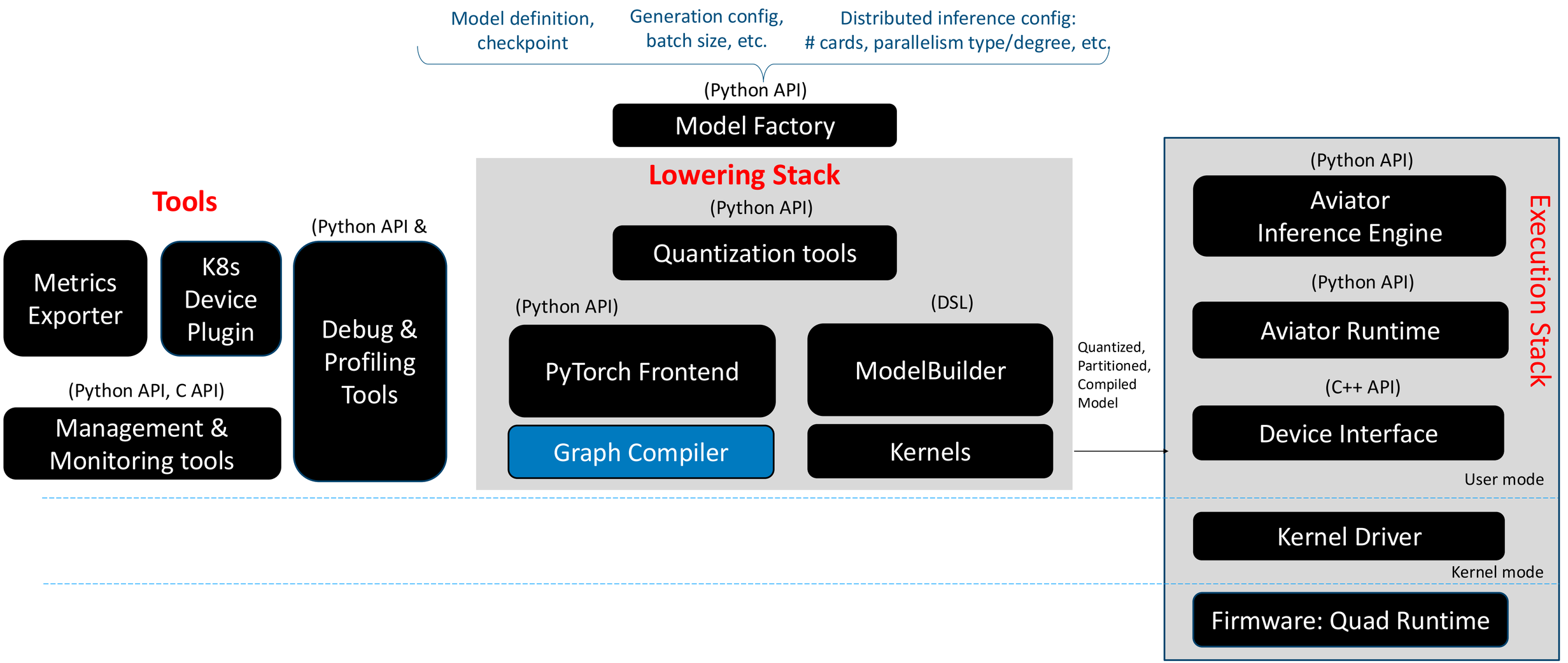
- Highest performance leverages fast, on-chip memory, closest to compute units
- Operations spread over chiplets/cards/servers/racks to aggregate compute and memory
- Parallelism strategy optimized for compute, memory, and communication costs

Corsair ISA and Graph Representation

Data movement	
MOVE_xxx2yyy	DMA based data transfer from - xxx: DDR, GM, ST, OB yyy: DDR, GM, ST, OB, WB, IB, DRE, SIMD
MOVE_xxx2PCIE	PCIe based peer-to-peer transfer
Compute	
MATMUL	Matrix multiplication using DIMC
CONV	Direct convolution using DIMC
EXEC_SIMD	Kernel function execution on SIMD cores
EXEC_CPU	Control function execution using gang CPU
Data reshape	
TRANSPOSE	Array transpose
EXTRACT	Rule-based sub-view extraction
INSERT	Rule-based sub-tensor insertion
Control	
LOOP	Nestable iterators
SEND/REC	CPU messaging via mailboxes
CONFIG_zzz	Pre-configure units (AC, SIMD, GM, ST)
BARRIER	Various barriers for stalling graph exec
RESET	Software triggered reset
NOP	Dummy, filler ops used for graph sync



Aviator Software Stack for Efficient, Scalable Inference



Graph Compiler for In-Memory Computing: Challenges

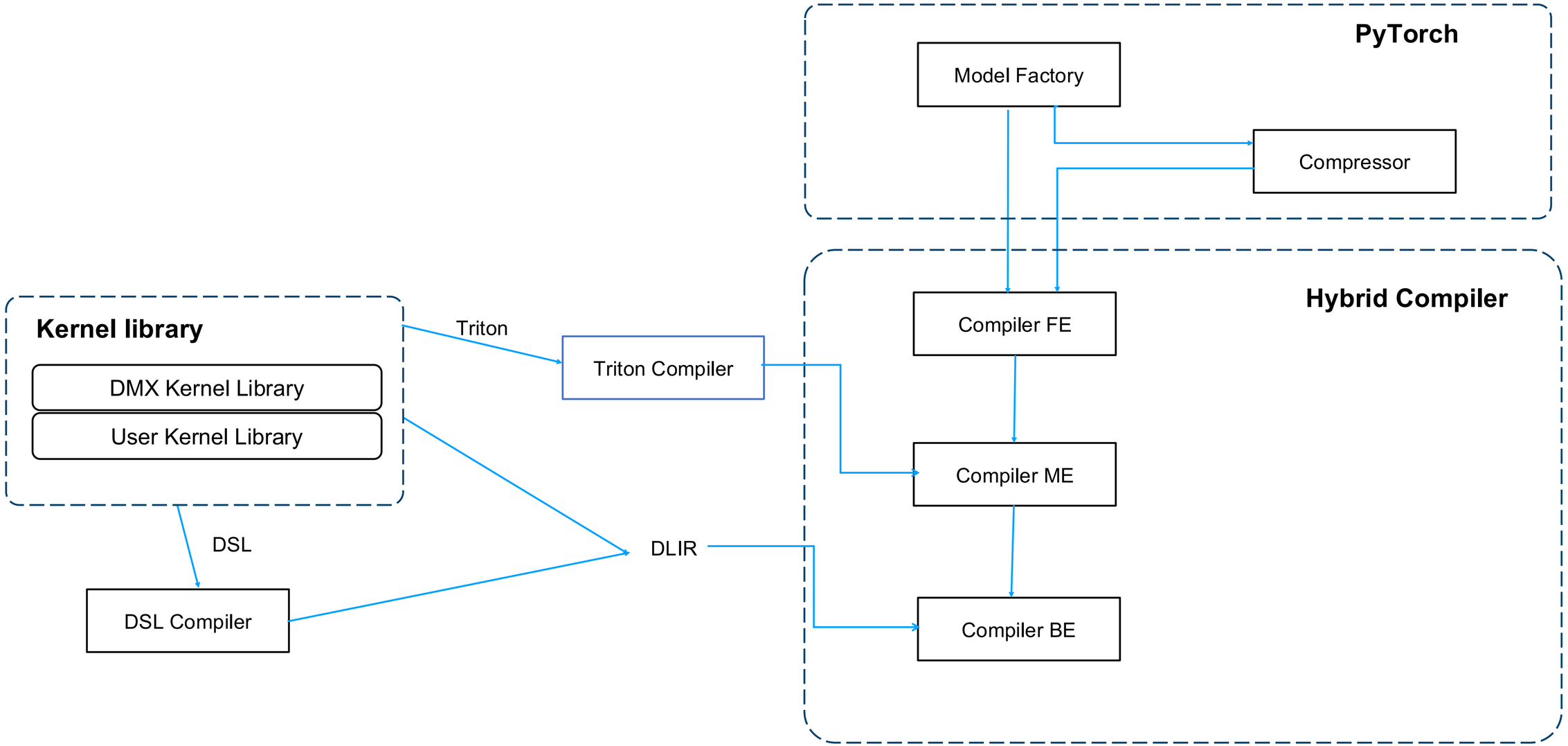
- Support of standard and custom ML operations
- Dynamic tensor shapes, conditions, autoregression
- Multi-device parallel model adaptation
- Tensor placement across memory hierarchy levels
- Fusion of large subgraphs into mega-kernels
- Resource sharing and lifetime management across multiple kernels
- Tiling, balance of spatial and temporal spread
- Stationarity, proximity, concurrency of operations

Ease, coverage of algorithm expression

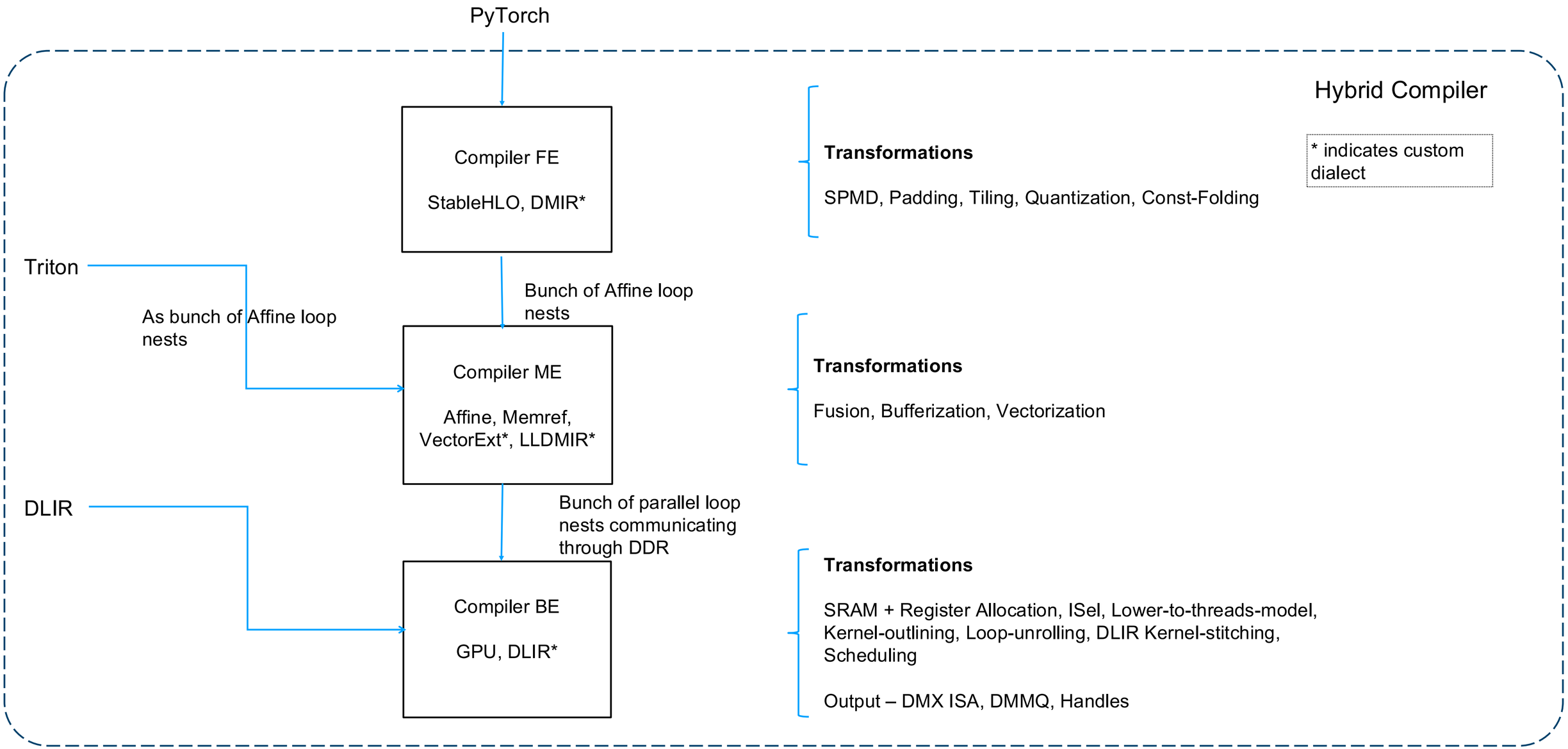
Developer intent and control

Hardware features and utilization

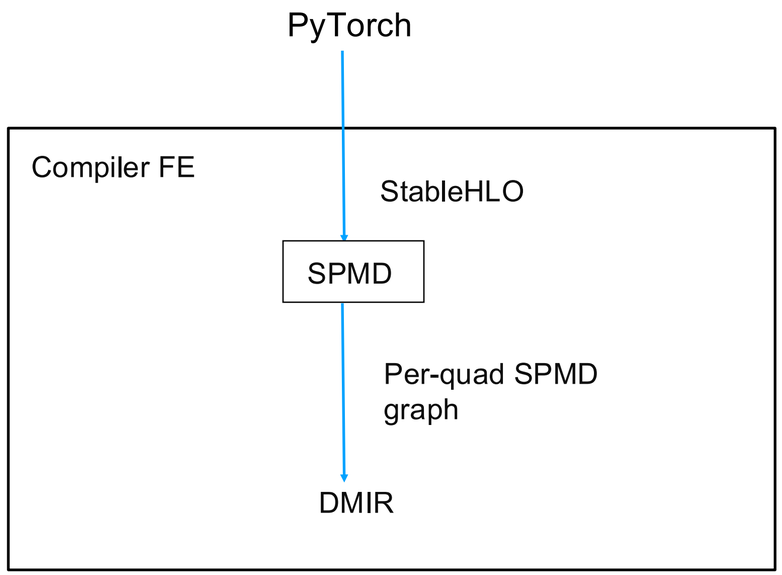
Aviator Lowering Stack: High Perf Kernels and MLIR Compiler



MLIR Graph Compiler Stack



SPMD partitioning



This per-quad symmetry maps quite nicely to multi-headed Attention models

Remember:
Card => 2 Packages
Package => 4 Chiplets
Chiplet => 4 quads

Total of 32 quads per card

Llama3 8b:
32 attention heads

GPT2 Med:
16 attention heads

whisper-medium:
16 attention heads

Progressive Lowering

A 512 x512 matmul through our compiler

```
module attributes {dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: tensor<512x512xf32>) -> tensor<512x512xf32> {
    %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512xf32>
    %1 = "dmir.matmul"(%arg0, %0) : (tensor<512x512xf32>, tensor<512x512xf32>) -> tensor<512x512xf32>
    return %1 : tensor<512x512xf32>
  }
}
```

```
func.func @main(%arg0: tensor<512x512x!dmir.block_fp<BFP16_64R>>) -> tensor<512x512xf16> {
  %c0 = arith.constant 0 : index
  %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512x!dmir.block_fp<BFP16_64C>>
  %1 = "dmir.undef"() : () -> tensor<512x512xf16>
  affine.for %arg1 = 0 to 8 {
    affine.for %arg2 = 0 to 8 {
      ...
      %7 = "dmir.matmul"(%extracted_slice, %extracted_slice_0) : (tensor<64x512x!dmir.block_fp<BFP16_64R>>, tensor<512x64x!dmir.block_fp<BFP16_64C>>) -> tensor<64x64x!dmir.block_fp<BFP32>>
      ...
    }
  }
  return %1 : tensor<512x512xf16>
}
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        %4 = "dmir.matmul"(%1, %3) : (!affine_ext.vec_ext<1x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, !affine_ext.vec_ext<8x1x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>)
          -> !affine_ext.vec_ext<1x1x!affine_ext.vec_ext<64x64xf16>>
        ...
      }
    }
  }
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        // 8 64x64 blocks of activation
        %18 = lldmir.matmul %1, %2, %3, %4, %5, %6, %7, %8 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>
        // 8 64x64 blocks of weights
        %10, %11, %12, %13, %14, %15, %16, %17 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>
        // Partial product reduction to calculate a single 64x64 block of output
        num_imc 8 ppr_mode 3 -> !affine_ext.vec_ext<64x64xf16>
        ...
      }
    }
    return
  }
}
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>], gpu.container_module} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %c8 = arith.constant 8 : index
    %c1 = arith.constant 1 : index
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    gpu.launch_func  @main_kernel::@main_kernel blocks in (%c1, %c1, %c1)
                                                 threads in (%c8, %c1, %c1)
                     args(%arg2 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg0 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %alloc : memref<8x8x!affine_ext.vec_ext<64x64xf16>>)

    return
  }

  gpu.module @main_kernel {
    gpu.func @main_kernel(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg1: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %arg2: memref<8x8x!affine_ext.vec_ext<64x64xf16>>) kernel attributes {known_block_size = array<i32: 8, 1, 1>, known_grid_size = array<i32: 1, 1, 1>} {...}
}
```

Progressive Lowering

A 512 x512 matmul through our compiler

```
module attributes {dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: tensor<512x512xf32>) -> tensor<512x512xf32> {
    %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512xf32>
    %1 = "dmir.matmul"(%arg0, %0) : (tensor<512x512xf32>, tensor<512x512xf32>) -> tensor<512x512xf32>
    return %1 : tensor<512x512xf32>
  }
}
```

```
func.func @main(%arg0: tensor<512x512x!dmir.block_fp<BFP16_64R>>) -> tensor<512x512xf16> {
  %c0 = arith.constant 0 : index
  %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512x!dmir.block_fp<BFP16_64C>>
  %1 = "dmir.undef"() : () -> tensor<512x512xf16>
  affine.for %arg1 = 0 to 8 {
    affine.for %arg2 = 0 to 8 {
      ...
      %7 = "dmir.matmul"(%extracted_slice, %extracted_slice_0) : (tensor<64x512x!dmir.block_fp<BFP16_64R>>, tensor<512x64x!dmir.block_fp<BFP16_64C>>) -> tensor<64x64x!dmir.block_fp<BFP32>>
      ...
    }
  }
  return %1 : tensor<512x512xf16>
}
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        %4 = "dmir.matmul"(%1, %3) : (!affine_ext.vec_ext<1x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, !affine_ext.vec_ext<8x1x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>)
          -> !affine_ext.vec_ext<1x1x!affine_ext.vec_ext<64x64xf16>>
        ...
      }
    }
  }
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        // 8 64x64 blocks of activation
        %18 = lldmir.matmul %1, %2, %3, %4, %5, %6, %7, %8 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>

        // 8 64x64 blocks of weights

        %10, %11, %12, %13, %14, %15, %16, %17 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>

        // Partial product reduction to calculate a single 64x64 block of output
        num_imc 8 ppr_mode 3 -> !affine_ext.vec_ext<64x64xf16>
        ...
      }
    }
    return
  }
}
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>], gpu.container_module} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %c8 = arith.constant 8 : index
    %c1 = arith.constant 1 : index
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    gpu.launch_func  @main_kernel::@main_kernel blocks in (%c1, %c1, %c1)
                                                 threads in (%c8, %c1, %c1)
                     args(%arg2 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg0 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %alloc : memref<8x8x!affine_ext.vec_ext<64x64xf16>>)

    return
  }

  gpu.module @main_kernel {
    gpu.func @main_kernel(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg1: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %arg2: memref<8x8x!affine_ext.vec_ext<64x64xf16>>) kernel attributes {known_block_size = array<i32: 8, 1, 1>, known_grid_size = array<i32: 1, 1, 1>} {...}
}
```

Progressive Lowering

A 512 x512 matmul through our compiler

```
module attributes {dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: tensor<512x512xf32>) -> tensor<512x512xf32> {
    %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512xf32>
    %1 = "dmir.matmul"(%arg0, %0) : (tensor<512x512xf32>, tensor<512x512xf32>) -> tensor<512x512xf32>
    return %1 : tensor<512x512xf32>
  }
}
```

Progressive Lowering

A 512 x512 matmul through our compiler

```
module attributes {dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: tensor<512x512xf32>) -> tensor<512x512xf32> {
    %0 = "dmir.const"() <{value = dense<"_____"> : tensor<512x512xf32>}> : () -> tensor<512x512xf32>
    %1 = "dmir.matmul"(%arg0, %0) : (tensor<512x512xf32>, tensor<512x512xf32>) -> tensor<512x512xf32>
    return %1 : tensor<512x512xf32>
  }
}
```

```
func.func @main(%arg0: tensor<512x512x!dmir.block_fp<BFP16_64R>>) -> tensor<512x512xf16> {
  %c0 = arith.constant 0 : index
  %0 = "dmir.const"() <{value = dense<"_____"> : tensor<512x512xf32>}> : () -> tensor<512x512x!dmir.block_fp<BFP16_64C>>
  %1 = "dmir.undef"() : () -> tensor<512x512xf16>
  affine.for %arg1 = 0 to 8 {
    affine.for %arg2 = 0 to 8 {
      ...
      %7 = "dmir.matmul"(%extracted_slice, %extracted_slice_0) : (tensor<64x512x!dmir.block_fp<BFP16_64R>>, tensor<512x64x!dmir.block_fp<BFP16_64C>>) -> tensor<64x64x!dmir.block_fp<BFP32>>
      ...
    }
  }
  return %1 : tensor<512x512xf16>
}
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        %4 = "dmir.matmul"(%1, %3) : (!affine_ext.vec_ext<1x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, !affine_ext.vec_ext<8x1x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>)
          -> !affine_ext.vec_ext<1x1x!affine_ext.vec_ext<64x64xf16>>
        ...
      }
    }
  }
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        // 8 64x64 blocks of activation
        %18 = lldmir.matmul %1, %2, %3, %4, %5, %6, %7, %8 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>
        // 8 64x64 blocks of weights

        %10, %11, %12, %13, %14, %15, %16, %17 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>

        // Partial product reduction to calculate a single 64x64 block of output
        num_imc 8 ppr_mode 3 -> !affine_ext.vec_ext<64x64xf16>
        ...
      }
    }
    return
  }
}
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>], gpu.container_module} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %c8 = arith.constant 8 : index
    %c1 = arith.constant 1 : index
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    gpu.launch_func  @main_kernel::@main_kernel blocks in (%c1, %c1, %c1)
                                                 threads in (%c8, %c1, %c1)
                     args(%arg2 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg0 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %alloc : memref<8x8x!affine_ext.vec_ext<64x64xf16>>)

    return
  }

  gpu.module @main_kernel {
    gpu.func @main_kernel(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg1: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %arg2: memref<8x8x!affine_ext.vec_ext<64x64xf16>>) kernel attributes {known_block_size = array<i32: 8, 1, 1>, known_grid_size = array<i32: 1, 1, 1>} {...}
}
```

Progressive Lowering

A 512 x512 matmul through our compiler

```
func.func @main(%arg0: tensor<512x512x!dmir.block_fp<BFP16_64R>>) -> tensor<512x512xf16> {
  %c0 = arith.constant 0 : index
  %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512x!dmir.block_fp<BFP16_64C>>
  %1 = "dmir.undef"() : () -> tensor<512x512xf16>
  affine.for %arg1 = 0 to 8 {
    affine.for %arg2 = 0 to 8 {
      ...
      %7 = "dmir.matmul"(%extracted_slice, %extracted_slice_0) : (tensor<64x512x!dmir.block_fp<BFP16_64R>>, tensor<512x64x!dmir.block_fp<BFP16_64C>>) -> tensor<64x64x!dmir.block_fp<BFP32>>
      ...
    }
  }
  return %1 : tensor<512x512xf16>
}
```

Progressive Lowering

A 512 x512 matmul through our compiler

```
module attributes {dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: tensor<512x512xf32>) -> tensor<512x512xf32> {
    %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512xf32>
    %1 = "dmir.matmul"(%arg0, %0) : (tensor<512x512xf32>, tensor<512x512xf32>) -> tensor<512x512xf32>
    return %1 : tensor<512x512xf32>
  }
}
```

```
func.func @main(%arg0: tensor<512x512x!dmir.block_fp<BFP16_64R>>) -> tensor<512x512xf16> {
  %c0 = arith.constant 0 : index
  %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512x!dmir.block_fp<BFP16_64C>>
  %1 = "dmir.undef"() : () -> tensor<512x512xf16>
  affine.for %arg1 = 0 to 8 {
    affine.for %arg2 = 0 to 8 {
      ...
      %7 = "dmir.matmul"(%extracted_slice, %extracted_slice_0) : (tensor<64x512x!dmir.block_fp<BFP16_64R>>, tensor<512x64x!dmir.block_fp<BFP16_64C>>) -> tensor<64x64x!dmir.block_fp<BFP32>>
      ...
    }
  }
  return %1 : tensor<512x512xf16>
}
```


```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        // 8 64x64 blocks of activation
        %18 = lldmir.matmul %1, %2, %3, %4, %5, %6, %7, %8 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>
        // 8 64x64 blocks of weights

        %10, %11, %12, %13, %14, %15, %16, %17 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>

        // Partial product reduction to calculate a single 64x64 block of output
        num_imc 8 ppr_mode 3 -> !affine_ext.vec_ext<64x64xf16>
        ...
      }
    }
    return
  }
}
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>], gpu.container_module} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %c8 = arith.constant 8 : index
    %c1 = arith.constant 1 : index
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    gpu.launch_func  @main_kernel::@main_kernel blocks in (%c1, %c1, %c1)
                                                 threads in (%c8, %c1, %c1)
                     args(%arg2 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg0 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %alloc : memref<8x8x!affine_ext.vec_ext<64x64xf16>>)

    return
  }

  gpu.module @main_kernel {
    gpu.func @main_kernel(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg1: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %arg2: memref<8x8x!affine_ext.vec_ext<64x64xf16>>) kernel attributes {known_block_size = array<i32: 8, 1, 1>, known_grid_size = array<i32: 1, 1, 1>} {...}
}
```

Progressive Lowering

A 512 x512 matmul through our compiler

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        %4 = "dmir.matmul"(%1, %3) : (!affine_ext.vec_ext<1x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, !affine_ext.vec_ext<8x1x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>)
          -> !affine_ext.vec_ext<1x1x!affine_ext.vec_ext<64x64xf16>>
        ...
      }
    }
}
```

Progressive Lowering

A 512 x512 matmul through our compiler

```
module attributes {dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: tensor<512x512xf32>) -> tensor<512x512xf32> {
    %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512xf32>
    %1 = "dmir.matmul"(%arg0, %0) : (tensor<512x512xf32>, tensor<512x512xf32>) -> tensor<512x512xf32>
    return %1 : tensor<512x512xf32>
  }
}
```

```
func.func @main(%arg0: tensor<512x512x!dmir.block_fp<BFP16_64R>>) -> tensor<512x512xf16> {
  %c0 = arith.constant 0 : index
  %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512x!dmir.block_fp<BFP16_64C>>
  %1 = "dmir.undef"() : () -> tensor<512x512xf16>
  affine.for %arg1 = 0 to 8 {
    affine.for %arg2 = 0 to 8 {
      ...
      %7 = "dmir.matmul"(%extracted_slice, %extracted_slice_0) : (tensor<64x512x!dmir.block_fp<BFP16_64R>>, tensor<512x64x!dmir.block_fp<BFP16_64C>>) -> tensor<64x64x!dmir.block_fp<BFP32>>
      ...
    }
  }
  return %1 : tensor<512x512xf16>
}
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        %4 = "dmir.matmul"(%1, %3) : (!affine_ext.vec_ext<1x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, !affine_ext.vec_ext<8x1x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>)
          -> !affine_ext.vec_ext<1x1x!affine_ext.vec_ext<64x64xf16>>
        ...
      }
    }
  }
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        // 8 64x64 blocks of activation
        %18 = lldmir.matmul %1, %2, %3, %4, %5, %6, %7, %8 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>

        // 8 64x64 blocks of weights

        %10, %11, %12, %13, %14, %15, %16, %17 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>

        // Partial product reduction to calculate a single 64x64 block of output
        num_imc 8 ppr_mode 3 -> !affine_ext.vec_ext<64x64xf16>
        ...
      }
    }
    return
  }
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>], gpu.container_module} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %c8 = arith.constant 8 : index
    %c1 = arith.constant 1 : index
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    gpu.launch_func  @main_kernel::@main_kernel blocks in (%c1, %c1, %c1)
                                                 threads in (%c8, %c1, %c1)
                     args(%arg2 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg0 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %alloc : memref<8x8x!affine_ext.vec_ext<64x64xf16>>)

    return
  }

  gpu.module @main_kernel {
    gpu.func @main_kernel(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg1: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %arg2: memref<8x8x!affine_ext.vec_ext<64x64xf16>>) kernel attributes {known_block_size = array<i32: 8, 1, 1>, known_grid_size = array<i32: 1, 1, 1>} {...}
}
```

Progressive Lowering

A 512 x512 matmul through our compiler

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        // 8 64x64 blocks of activation
        %18 = lldmir.matmul %1, %2, %3, %4, %5, %6, %7, %8 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>
        // 8 64x64 blocks of weights

        %10, %11, %12, %13, %14, %15, %16, %17 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>

        // Partial product reduction to calculate a single 64x64 block of output
        num_imc 8 ppr_mode 3 -> !affine_ext.vec_ext<64x64xf16>
        ...
      }
    }
    return
  }
}
```

Progressive Lowering

A 512 x512 matmul through our compiler

```
module attributes {dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: tensor<512x512xf32>) -> tensor<512x512xf32> {
    %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512xf32>
    %1 = "dmir.matmul"(%arg0, %0) : (tensor<512x512xf32>, tensor<512x512xf32>) -> tensor<512x512xf32>
    return %1 : tensor<512x512xf32>
  }
}
```

```
func.func @main(%arg0: tensor<512x512x!dmir.block_fp<BFP16_64R>>) -> tensor<512x512xf16> {
  %c0 = arith.constant 0 : index
  %0 = "dmir.const"() <{value = dense<"..."> : tensor<512x512xf32>}> : () -> tensor<512x512x!dmir.block_fp<BFP16_64C>>
  %1 = "dmir.undef"() : () -> tensor<512x512xf16>
  affine.for %arg1 = 0 to 8 {
    affine.for %arg2 = 0 to 8 {
      ...
      %7 = "dmir.matmul"(%extracted_slice, %extracted_slice_0) : (tensor<64x512x!dmir.block_fp<BFP16_64R>>, tensor<512x64x!dmir.block_fp<BFP16_64C>>) -> tensor<64x64x!dmir.block_fp<BFP32>>
      ...
    }
  }
  return %1 : tensor<512x512xf16>
}
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        %4 = "dmir.matmul"(%1, %3) : (!affine_ext.vec_ext<1x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, !affine_ext.vec_ext<8x1x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>)
          -> !affine_ext.vec_ext<1x1x!affine_ext.vec_ext<64x64xf16>>
        ...
      }
    }
  }
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>]} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    affine.for %arg3 = 0 to 8 {
      affine.for %arg4 = 0 to 8 {
        ...
        // 8 64x64 blocks of activation
        %18 = lldmir.matmul %1, %2, %3, %4, %5, %6, %7, %8 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>,
                                                             !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>

        // 8 64x64 blocks of weights

        %10, %11, %12, %13, %14, %15, %16, %17 : !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>,
                                                 !affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>

        // Partial product reduction to calculate a single 64x64 block of output
        num_imc 8 ppr_mode 3 -> !affine_ext.vec_ext<64x64xf16>
        ...
      }
    }
    return
  }
}
```

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>], gpu.container_module} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %c8 = arith.constant 8 : index
    %c1 = arith.constant 1 : index
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    gpu.launch_func  @main_kernel::@main_kernel blocks in (%c1, %c1, %c1)
                                                 threads in (%c8, %c1, %c1)
                     args(%arg2 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg0 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %alloc : memref<8x8x!affine_ext.vec_ext<64x64xf16>>)

    return
  }

  gpu.module @main_kernel {
    gpu.func @main_kernel(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg1: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %arg2: memref<8x8x!affine_ext.vec_ext<64x64xf16>>) kernel attributes {known_block_size = array<i32: 8, 1, 1>, known_grid_size = array<i32: 1, 1, 1>} {...}
```

Progressive Lowering

A 512 x512 matmul through our compiler

```
module attributes {dm.model_constants.type = [tensor<512x512x!dmir.block_fp<BFP16_64C>>], dm.model_inputs.type = [tensor<512x512xf32>], dm.model_outputs.type = [tensor<512x512xf32>], gpu.container_module} {
  func.func @main(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>, %arg1: memref<8x8x!affine_ext.vec_ext<64x64xf16>>, %arg2: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>) {
    %c8 = arith.constant 8 : index
    %c1 = arith.constant 1 : index
    %alloc = memref.alloc() : memref<8x8x!affine_ext.vec_ext<64x64xf16>>
    gpu.launch_func  @main_kernel::@main_kernel blocks in (%c1, %c1, %c1)
                                                 threads in (%c8, %c1, %c1)
                     args(%arg2 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg0 : memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %alloc : memref<8x8x!affine_ext.vec_ext<64x64xf16>>)

    return
  }

  gpu.module @main_kernel {
    gpu.func @main_kernel(%arg0: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64C>>>,
                          %arg1: memref<8x8x!affine_ext.vec_ext<64x64x!dmir.block_fp<BFP16_64R>>>,
                          %arg2: memref<8x8x!affine_ext.vec_ext<64x64xf16>>) kernel attributes {known_block_size = array<i32: 8, 1, 1>, known_grid_size = array<i32: 1, 1, 1>} {...}
}
```

Number of Affine Dialect Based Optimizations b/w FE/ME and ME/BE

FE/ME

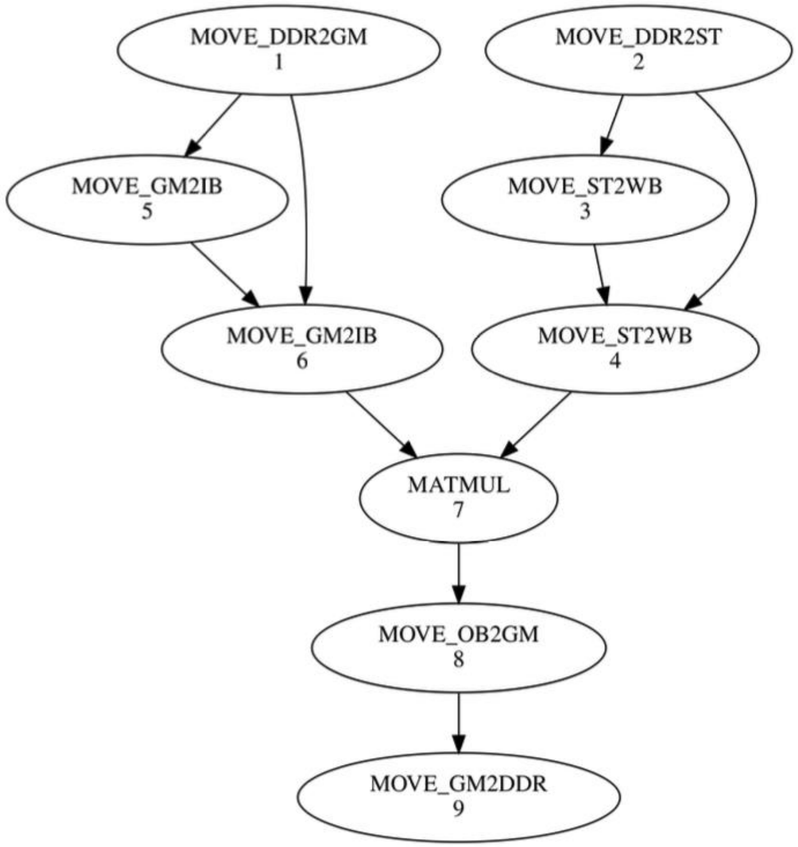
1. Loop permutation – Make loops w/ highest trip counts outermost
2. Single trip loop promotion – Promote single iteration loops (inner-most only)
3. Normalization – Normalize loop nests to enable producer-consumer fusion

ME/BE

1. Affine scalar replacement – Forward affine memref stores to loads, remove redundant loads, remove dead stores

DLIR: MLIR Egress to DMX ISA

```
"dlir.dmmq"() <{sym_name = "scmm"}> ({
  "dlir.scheduled_graph"() <{sym_name = "scmm"}> ({
    "dlir.dispatch_queue"() <{sym_name = "queue_0"}> ({

      %1 = "dlir.task_group"(%0) ({
        ...
        // DDR2GM
        %9 = "dlir.dma_config"(...)
        %10 = "dlir.copy"(%9, ...)

        // DDR2ST
        %11 = "dlir.dma_config"(...)
        %12 = "dlir.copy"(%11, ...)
      }) : (index) -> !dlir.token


      %3 = "dlir.task_group"(%2) ({
        ...

        // GM2IB
        %13 = "dlir.dma_config"(...)
        %14 = "dlir.copy"(%13, ...)

        // ST2WB
        %15 = "dlir.dma_config"(...)
        %16 = "dlir.copy"(%15, ...)


        %17 = "dlir.task_barrier"(%14, %16) : (index) -> !dlir.token

        // Produces result in OB
        %18 = "dlir.matmul"(...)

        %19 = "dlir.task_barrier"(%18) : (index) -> !dlir.token

        // OB2GM
        %20 = "dlir.dma_config"(...)
        %21 = "dlir.copy"(%20, ...)

        %22 = "dlir.task_barrier"(%21) : (index) -> !dlir.token

        // GM2DDR
        %23 = "dlir.dma_config"(...)
        %24 = "dlir.copy"(%23, ...)

      }) : (!dlir.token) -> !dlir.token
    }) : () -> ()
  }) : () -> ()
```



Conclusion

MLIR enabled us to build an ergonomic S/W stack that we're confident can realize the full potential of our hardware.

- Compatibility with PyTorch allowed us to meet users where they are.
- Flexibility and extensible dialect system -
  - Allowed us to faithfully model and thereby optimize for our hardware and numerics, via custom IRs.
  - Allowed us to support both code-gen and kernel-native approaches via a single compiler.

Next steps

- Op coverage
- Performance
- Kernel libraries