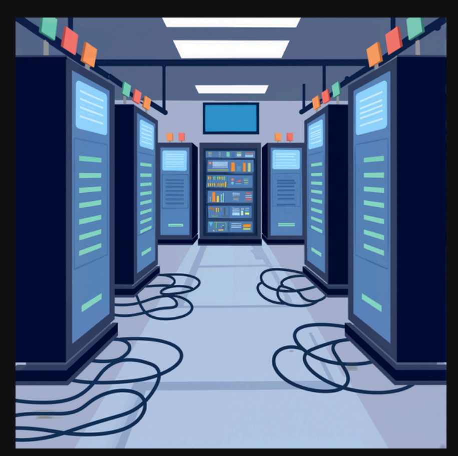
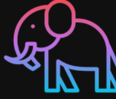


TASK-BASED MLIR COMPILATION FOR HPC CLUSTERS

Robert K Samuel CS23S042

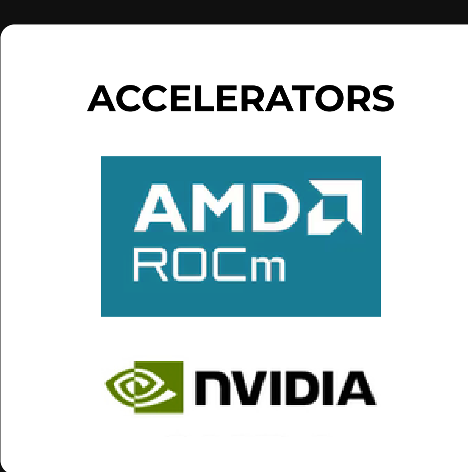
Indian Institute of Technology, Madras

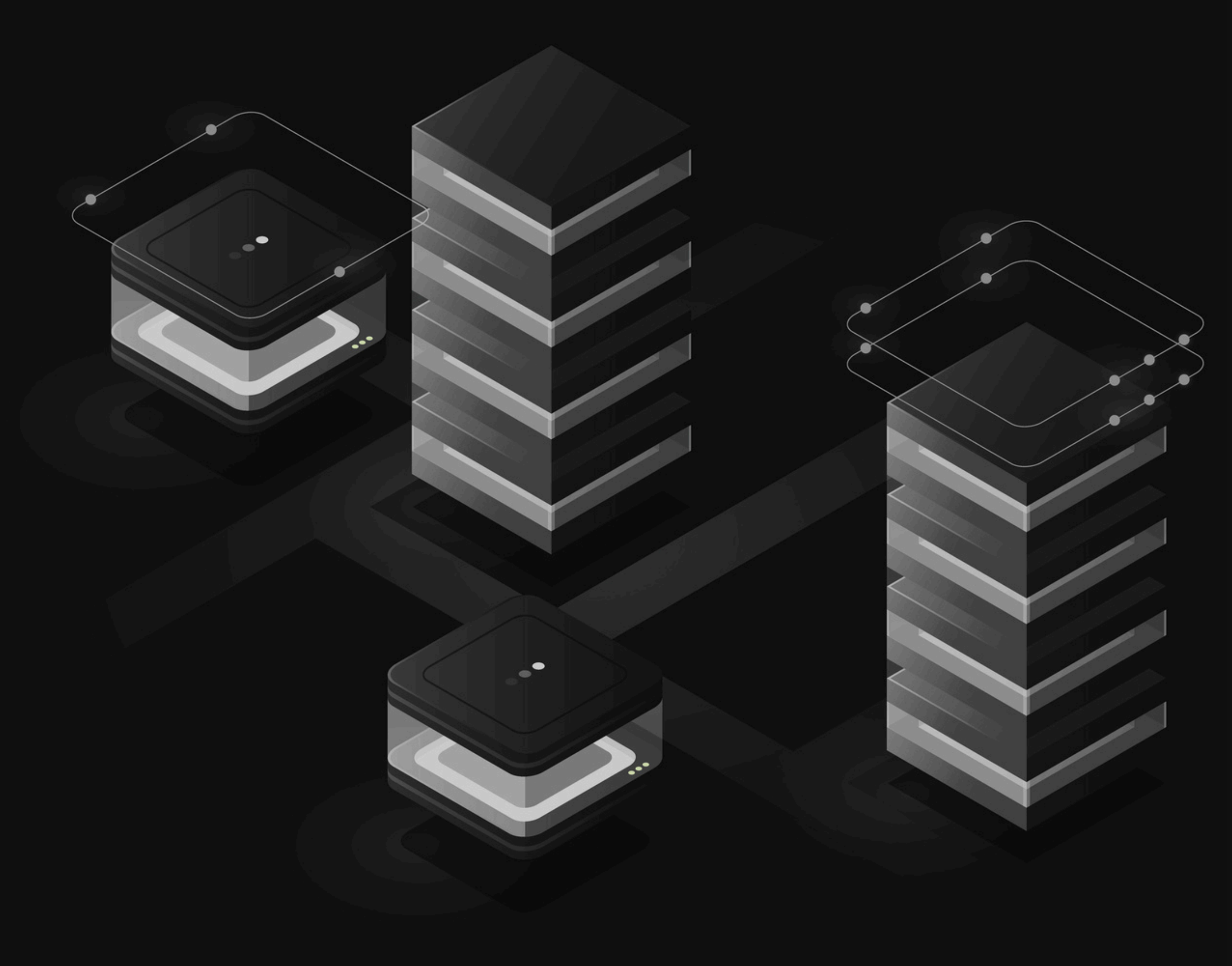


THE HPC COMPLEXITY WALL

Efficiently programming a cluster requires manually integrating disjoint programming models. A simple algorithm gets lost in the intricacies of data partitioning, synchronization, and race condition management.

- Data Partitioning
- Synchronization
- Race condition





THE CURRENT TRADE-OFFS IN HPC COMPILATION

Existing approaches leave a gap between programmability and system level heterogeneity.

- CPU Only
  3 Nodes, 96 Cores Total
- GPU Only
  3 Nodes, 3 GPUs
- Heterogeneous
  3 Nodes, 96 Cores, 3 GPUs

Domain-Specific Languages

+ High Productivity
– Learn new language
– Tied to specific domains
(e.g., StarPlat for Graph analytics)

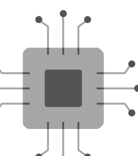
Libraries

+ Optimized by experts
+ Existing language
– Cannot customize
(e.g., TensorFlow for ML)

Manual Compilation

+ Full control
– Manual Optimizations
– Error-prone & Complex

Retargetable compilation



+ 0 Code Rewrite
+ Generate code for different hardwares

THE MLIR

The LEGO of compiler infrastructure

THE "SEMANTIC GAP" PROBLEM

- Lose all the high-level intent
- Mess of pointers and assembly-like instructions
- High-level operations in a Dialect
- Progressive Lowering

triangle_count.starplat

```
function Compute_TC(Graph g) {
 long triangle_count = 0;
 forall(v in g.nodes()) {
  forall(u in g.neighbors(v)) {
   forall(w in g.neighbors(v))
    if (g.is_an_edge(u, w)) {
     triangle_count += 1;
     }
    }
   }
 }
 return triangle_count;
}
```

AST

FunctionDeclaration
Parameter
Return Type
Body
Graph
"g"
long
blck stmt
...

triangle_count.starplat_mlir

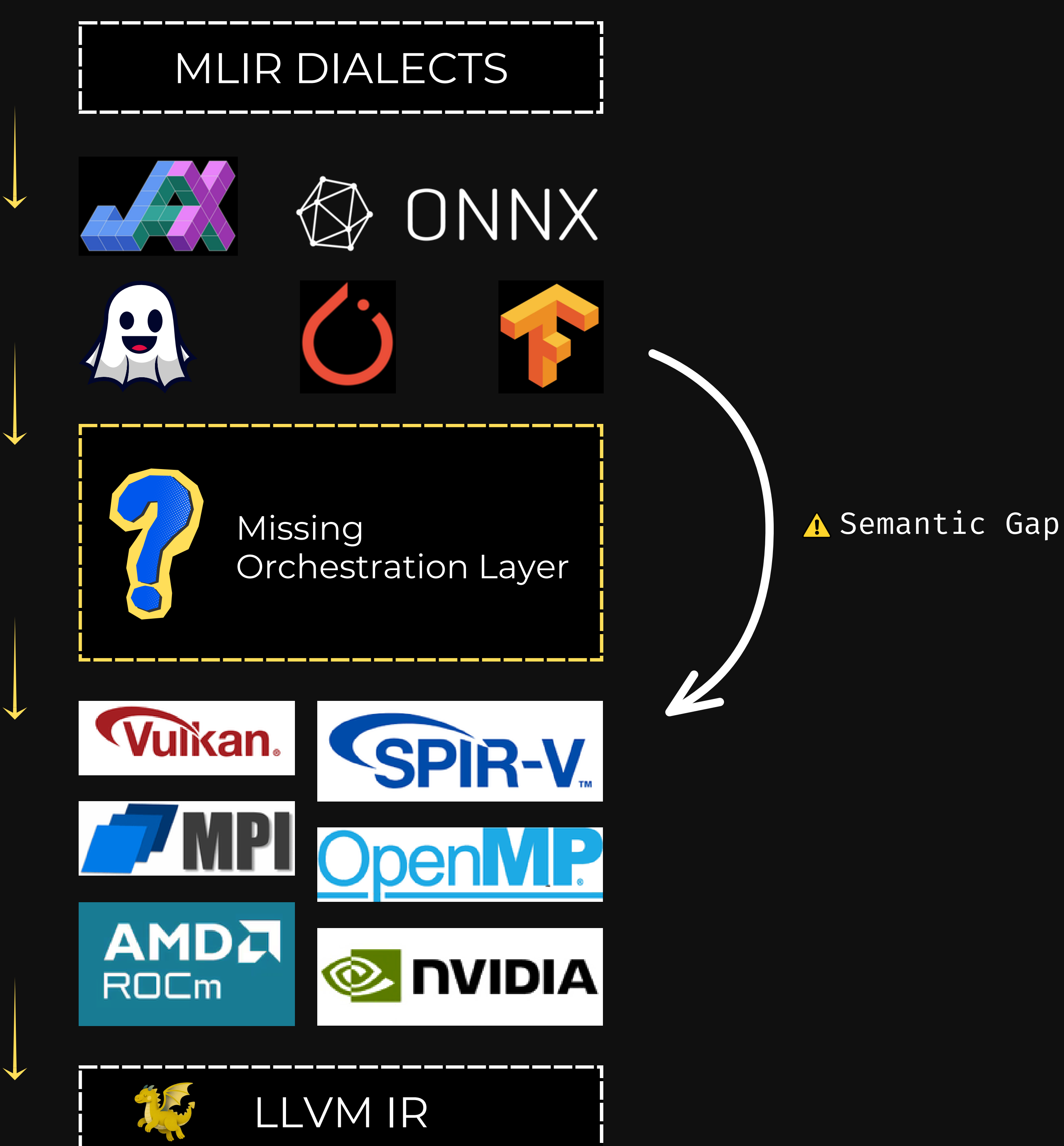
```
func.func @Compute_TC(%g : !starplat.graph) -> i64 {
  %0 = arith.constant 0
  %triangle_count = memref.alloca i64
  starplat.forall (%v) in (starplat.nodes %g) {
    starplat.forall (%u) in (starplat.neighbors %g, %v) {
      starplat.forall (%w) in (starplat.neighbors %g, %v) {
        %edge_exists = starplat.is_edge %g, %u, %w
        starplat.if %edge_exists {
          %old = memref.load %triangle_count
          %one = arith.constant 1
          %new = arith.add %old, %one
          memref.store %new, %triangle_count
        }
      }
    }
  }
  %ret = memref.load %triangle_count
  func.return %ret
}
```

THE GAP____ IN THE COMPILATION STACK

No existing dialect bridges high-level task parallelism with distributed multi-device execution

WE NEED AN IR THAT UNDERSTANDS

- Where does a task run?
- Are the tasks dependent?
- How does it communicate?
- Is there Intra-Task parallelism?



THE GAP DHIR IN THE COMPILATION STACK

A Task-based intermediate representation designed explicitly for HPC clusters

DHIR IS AN IR THAT UNDERSTANDS

- Are tasks dependent?
- Where a task run?
- How it communicates?
- Intra-Task parallelism?

MLIR DIALECTS

Semantic Bridge

MOTIVATING EXAMPLE

A Step by Step Transformation of Matrix Multiplication in DHIR

- The actual kernel remains untouched

matmul.c

```c
int matmul(int *A, int *B, int *C, int N)
{
  for (int i = 0; i < N; ++i) {
   for (int j = 0; j < N; ++j) {
    C[i*N + j] = 0;
    for (int k = 0; k < N; ++k) {
     C[i*N + j] += A[i*N + k] * B[k*N + j];
    }
   }
  }
 return 0;
}
```

matmul_affine.mlir

```
affine.for %i = 0 to N {
 affine.for %j = 0 to N {
 affine.store %c0, %arg2[%i, %j] : memref<?x?xf32>
  affine.for %k = 0 to N {
   %a = affine.load %A[%i, %k] : memref<?x?xf32>
   %b = affine.load %B[%k, %j] : memref<?x?xf32>
   %c = affine.load %C[%i, %j] : memref<?x?xf32>
   %prod = arith.mulf %a, %b : f32
   %sum = arith.addf %c, %prod : f32
   affine.store %sum, %C[%i, %j]
  }
 }
}
```

matmul_dhir.mlir

```
...
dhir.schedule() <{inputs = [
 {name = "arg0", type = i32},
 {name = "arg1", type = memref<?x?xf32>},
 {name = "arg2", type = memref<?x?xf32>},
 {name = "arg3", type = memref<?x?xf32>}],
 scheduleName = "matmul"}> ({
  ...
  dhir.partition(%arg0, %arg1, %arg2, %agr3)
  <{operandSegmentSizes = array<i32: 2, 1>}> ({
    affine.for %i = 0 to %arg0 {
     affine.for %j = 0 to %arg0 {
      affine.store %c0, %arg2[%i, %j] : memref<?x?xf32>
      affine.for %k = 0 to %arg0 {
       %2 = affine.load %arg0[%i, %k] : memref<?x?xf32>
       %3 = affine.load %arg1[%k, %j] : memref<?x?xf32>
       %4 = affine.load %arg2[%i, %j] : memref<?x?xf32>
       %5 = arith.mulf %2, %3 : f32
       %6 = arith.addf %4, %5 : f32
       affine.store %6, %arg2[%i, %j] : memref<?x?xf32>
   } } } dhir.return() })
   dhir.return()
})
...
```

LOWERING TO DHIR PRIMITIVES

Decomposing High Level Matrix Operations into Core DHIR Constructs

dhir::ScheduleOp

matmul_dhir_primitive.mlir

```
dhir.schedule() {
...
%3 = dhir.task(%arg4, %subview, %subview_0)
<{inpRanges = array<i64: 0, 0>,
operandSegmentSizes = array<i32: 2, 1, 1>,
outRanges = array<i64: 0, %o2>,
target = #dlti.target_device_spec<
    "type" = "node",
    "arch" = "x86_64",
    "cost" = 0.500000e+00 : f32,
    "node_id" = "node0",
    "gpu_count" = 0 : i32,
    "gpu_arch" = [],
    "gpu_cc" = []
    "gpu_id" = []>}>
({
...
scf.parallel (%arg9) = (%c0) to (%cn3) step (%c1) {
 ...
 scf.for %arg10 = %c0 to %cn step %c1 {
  memref.store %cst, %subview_0[%arg9, %arg10]
  ...
  scf.for %arg11 = %c0 to %cn step %c1 {
    %6 = memref.load %subview[%arg9, %arg11]
    %7 = memref.load %arg4[%arg11, %arg10]
    %8 = arith.mulf %6, %7 : f32
    %9 = memref.load %subview_0[%arg9, %arg10]
    %10 = arith.addf %9, %8 : f32
    memref.store %10, %subview_0[%arg9, %arg10]
    }
 }
 scf.reduce
}
dhir.return() : () -> ()
}) {needBroadcast = false} : () -> !dhirl.taskref

%4 = dhir.task(%arg3, %arg4, %arg5)
<{inpRanges = array<i64: 0, 0>,
operandSegmentSizes = array<i32: 0, 1, 1>,
outRanges = array<i64: %co2, %co3>,
target = #dlti.target_device_spec<
    "type" = "node",
    "arch" = "x86_64",
    "cost" = 0.500000e-01 : f32,
    "node_id" = "node1",
    "gpu_count" = 1 : i32,
    "gpu_arch" = ["pascal"],
    "gpu_cc" = ["sm_61"]
    "gpu_id" = [0 : i32]>}>
    ({
        ...
    }
  }
 }
 dhir.return() : () -> ()
 }) {needBroadcast = false} : () -> !dhir.taskref
 ...
}
```

Arguments

Target Information

Task Body

Input Attributes

Attributes

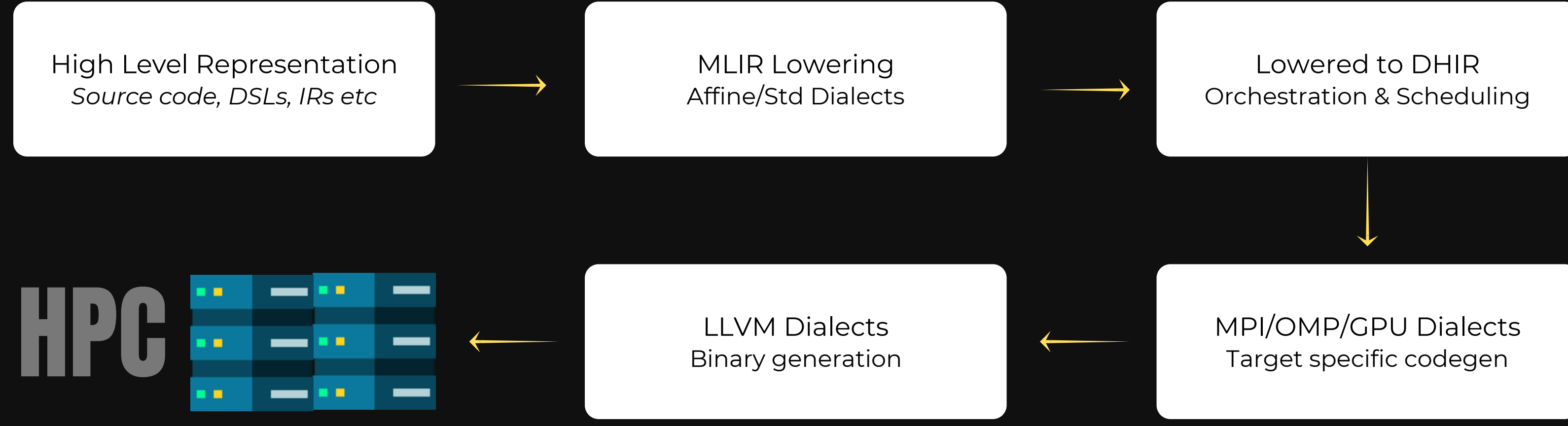
system_config.json


```
{
  "cluster": {
    "world_size": 3,
    "node_ids": ["node0", "node1", "node2"]
  },

  "nodes": {
    "node0": {
      "cpu_arch": "x86_64", "cost" : 0.5,
      "gpus": [

      ]
    },
    "node1": {
      "cpu_arch": "x86_64", "cost" : 0.5,
      "gpus": [
        { "id": 0, "cc": "sm_61", "arch":"pascal" }
      ]
    }
  }
}
```

BACKEND SPECIFIC IR GENERATION

Lowering DHIR to Target Aware
Intermediate Representation

OVERALL DHIR COMPILATION FLOW



```
%0 = mpi.init : !mpi.retval
%1 = mpi.comm_world : !mpi.comm
%rank = mpi.comm_rank(%1) : i32
%retval, %size = mpi.comm_size(%1) : !mpi.retval, i32
%cst = arith.constant 0.000000e+00 : f32
%2 = arith.index_cast %arg0 : i32 to index
%3 = arith.index_cast %arg1 : i32 to index
%4 = arith.index_cast %arg2 : i32 to index
%c0_i32 = arith.constant 0 : i32
%5 = arith.remsi %c0_i32, %size : i32
%6 = arith.cmpi eq, %rank, %5 : i32
scf.if %6 {
  %c15 = arith.constant 15 : index
  %c30 = arith.constant 30 : index
  %c1 = arith.constant 1 : index
  omp.parallel {
   omp.wsloop {
     omp.loop_nest (%arg6) : index = (%c15) to (%c30) step (%c1){
      %c0 = arith.constant 0 : index
      %c0_0 = arith.constant 0 : index
      %c20 = arith.constant 20 : index
      %c1_1 = arith.constant 1 : index
      scf.for %arg7 = %c0_0 to %c20 step %c1_1 {
      memref.store %cst, %arg5[%c0, %arg7] : memref<?x128xf32>
      %c0_2 = arith.constant 0 : index
      %c10 = arith.constant 10 : index
      %c1_3 = arith.constant 1 : index
      scf.for %arg8 = %c0_2 to %c10 step %c1_3 {
        %10 = memref.load %arg3[%c0, %arg8] : memref<?x128xf32>
        %11 = memref.load %arg4[%arg8, %arg7] : memref<?x128xf32>
        %12 = arith.mulf %10, %11 : f32
        %13 = memref.load %arg5[%c0, %arg7] : memref<?x128xf32>
        %14 = arith.addf %13, %12 : f32
        memref.store %14, %arg5[%c0, %arg7] : memref<?x128xf32>
        }
      }
     omp.yield
    }
   }
   omp.terminator
  }
 }
 %c1_i32 = arith.constant 1 : i32
 %7 = arith.remsi %c1_i32, %size : i32
 %8 = arith.cmpi eq, %rank, %7 : i32
 scf.if %8 {
 %c0 = arith.constant 0 : index
 %c15 = arith.constant 15 : index
 %c1 = arith.constant 1 : index
 ...
 gpu.launch_func  @kernel_matmul(...)
 ...
 }
 %9 = mpi.barrier(%1) -> !mpi.retval
```

CORE ABSTRACTIONS

Introducing core DHIR operations

CORE ABSTRACTIONS

dhir::TaskOp

Encapsulates a region of code that can be independently scheduled

THE UNIT OF WORK

- Explicit Memory Access
- Target Mapping
- Collective Flags

matmul_affine.mlir

```
%0 = dhir.task(%arg4, %subview, %subview_0)
<{inpRanges = array<i64: 0, 0>,
operandSegmentSizes = array<i32: 2, 1, 1>,
outRanges = array<i64: 0, %o2>,
target = #dlti.target_device_spec<
 "type" = "node",
 "arch" = "x86_64",
 "cost" = 1.000000e+00 : f32,
 "node_id" = "node0",
 "gpu_count" = 0 : i32,
 "gpu_arch" = [],
 "gpu_cc" = []
 "gpu_id" = []>}>
({
    ...
    dhir.return() : () -> ()
}) {needBroadcast = false} : () -> !dhirl.taskref
```

Memory Accesses

Target Mapping

Task Body

Collective Flag

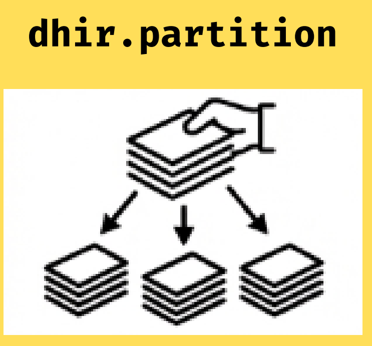
CORE ABSTRACTIONS

dhir::PartitionOp

Expands loop nests into
multiple task operations

THE DISTRIBUTOR

- Dependence-Free Loop Nest
- Hardware Config File
- Distribution



matmul_dhir.mlir

```
...
dhir.schedule() <{inputs = [
 {name = "arg0", type = i32},
 {name = "arg1", type = memref<?x?xf32>},
 {name = "arg2", type = memref<?x?xf32>},
 {name = "arg3", type = memref<?x?xf32>}],
 scheduleName = "matmul"}> ({
  ...
  dhir.partition(%arg0, %arg1, %arg2, %agr3)
  <{operandSegmentSizes = array<i32: 2, 1>}> ({
    affine.for %i = 0 to %arg0 {
     affine.for %j = 0 to %arg0 {
      affine.store %c0, %arg2[%i, %j] : memref<?x?xf32>
      affine.for %k = 0 to %arg0 {
       %2 = affine.load %arg0[%i, %k] : memref<?x?xf32>
       %3 = affine.load %arg1[%k, %j] : memref<?x?xf32>
       %4 = affine.load %arg2[%i, %j] : memref<?x?xf32>
       %5 = arith.mulf %2, %3 : f32
       %6 = arith.addf %4, %5 : f32
       affine.store %6, %arg2[%i, %j] : memref<?x?xf32>
   } } } dhir.return() })
   dhir.return()
})
...
```

→

matmul_dhir_primitive.mlir

```
dhir.schedule() {
...
%3 = dhir.task(%arg4, %subview, %subview_0)
<{inpRanges = array<i64: 0, 0>,
operandSegmentSizes = array<i32: 2, 1, 1>,
outRanges = array<i64: 0, %o2>,
target = #dlti.target_device_spec<
    "type" = "node",
    "arch" = "x86_64",
    "cost" = 1.000000e+00 : f32,
    "node_id" = "node0",
    "gpu_count" = 0 : i32,
    "gpu_arch" = [],
    "gpu_cc" = []
    "gpu_id" = []>}>
({
...
scf.parallel (%arg9) = (%c0) to (%cn3) step (%c1) {
 ...
 scf.for %arg10 = %c0 to %cn step %c1 {
  memref.store %cst, %subview_0[%arg9, %arg10]
  ...
  scf.for %arg11 = %c0 to %cn step %c1 {
    %6 = memref.load %subview[%arg9, %arg11]
    %7 = memref.load %arg4[%arg11, %arg10]
    %8 = arith.mulf %6, %7 : f32
    %9 = memref.load %subview_0[%arg9, %arg10]
    %10 = arith.addf %9, %8 : f32
    memref.store %10, %subview_0[%arg9, %arg10]
    }
 }
 scf.reduce
}
dhir.return() : () -> ()
}) {needBroadcast = false} : () -> !dhirl.taskref

%4 = dhir.task(%arg3, %arg4, %arg5)
<{inpRanges = array<i64: 0, 0>,
operandSegmentSizes = array<i32: 0, 1, 1>,
outRanges = array<i64: %co2, %co3>,
target = #dlti.target_device_spec<
    "type" = "node",
    "arch" = "x86_64",
    "cost" = 5.000000e-01 : f32,
    "node_id" = "node1",
    "gpu_count" = 1 : i32,
    "gpu_arch" = ["pascal"],
    "gpu_cc" = ["sm_61"]
    "gpu_id" = [0 : i32]>}>
    ({
        ...
    }
  }
 }
 dhir.return() : () -> ()
 }) {needBroadcast = false} : () -> !dhir.taskref
 ...
}
```

CORE ABSTRACTIONS

`dhir::ScheduleOp`

The '*Main Function*' of the distributed system

THE ORCHESTRATOR

- Top-level construct
- Coordinate execution across the cluster.
- Global Scope
- Memory Regions

matmul_affine.mlir

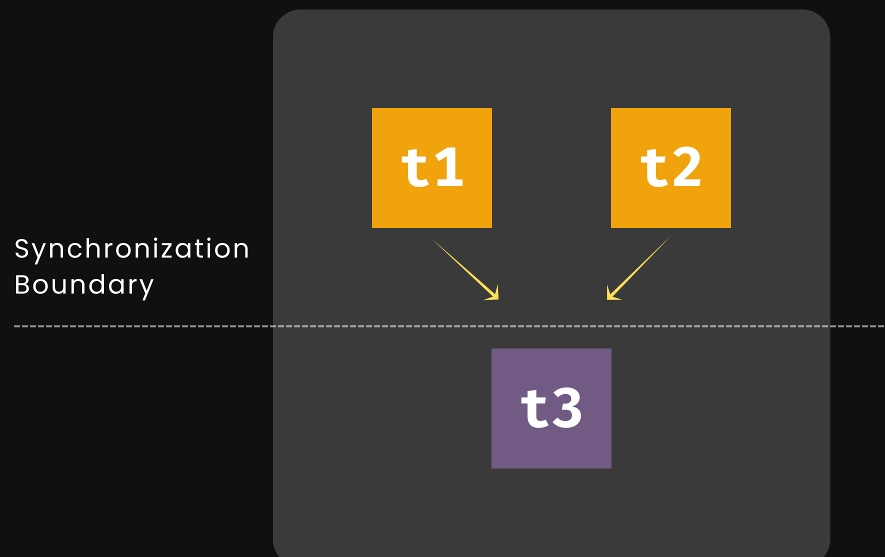
```
dhir.schedule() <{inputs = [
 {name = "arg0", type = i32},
 {name = "arg1", type = memref<?x?xf32>},
 {name = "arg2", type = memref<?x?xf32>},
 {name = "arg3", type = memref<?x?xf32>}],
 scheduleName = "matmul"}> ({
  ...
   dhir.return()
})
```

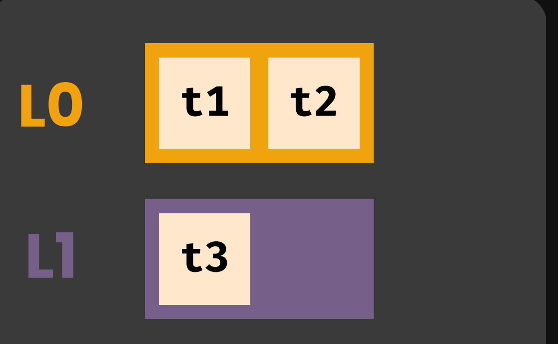
Global Memory

Global Scope

THE EXECUTION MODEL

Tasks within the same level are guaranteed to be safe for parallel execution





DISTRIBUTION MECHANISM

- DAG based on ins/outs
- Tasks within same level
- Non aliasing assumption
- Semantically equivalent to sequential execution

sample_dhir.mlir

```
dhir.schedule %a, %b, %c, %d, %e {
%t1 = dhir.task %cpu, %a, %b, %c {
 scf.for %i = 0 to 20 {
  %x = memref.load %a[%i]
  %y = memref.load %b[%i]
  %z = arith.addi %x, %y
  memref.store %z, %c[%i]
  }
 }

%t2 = dhir.task %cpu, %a, %b, %d {
 scf.for %i = 0 to 20 {
  %x = memref.load %a[%i]
  %y = memref.load %b[%i]
  %z = arith.addi %x, %y
  memref.store %z, %d[%i]
  }
 }

%t3 = dhir.task %gpu, %c, %d, %e {
 scf.for %i = 0 to 20 {
 %x = memref.load %c[%i]
 %y = memref.load %d[%i]
 %z = arith.addi %x, %y
 memref.store %z, %e[%i]
  }
 }

}
```

ins outs

CONVERSIONS

Introducing DHIR conversions

AFFINE-TO-DHIR LOWERING

Analyzes affine loop nests and determines whether partitioning is appropriate

LOOP DEPENDENCE ANALYSIS

- **Index Set Construction**
  Builds constraints from loop nest hierarchy
- **All-Pairs Dependence Check**
  Analyzes every memory operation pair
- **MLIR Affine Analysis**
  Uses ***checkMemrefAccessDependence*** with depth=1

```
Domain: 0, Range: 3, Symbols: 0,
Locals: 0

( ) -> (
Id<0x61bd0476df50>
Id<0x61bd0476e180>
Id<0x61bd0476ee40> ) :
[ ]6 constraints

(Value  Value   Value   const)
  1     0     0     0   >= 0
 -1     0     0     999 >= 0
  0     1     0     0   >= 0
  0    -1     0     999 >= 0
  0     0     1     0   >= 0
  0     0    -1     999 >= 0
```

Constraint Matrix

matmul_affine.mlir

```
affine.for %i = 0 to 1000 {
 affine.for %j = 0 to 1000 {
  affine.for %k = 0 to 1000 {
   %a = affine.load %A[%i, %k] : memref<?x?xf32>
   %b = affine.load %B[%k, %j] : memref<?x?xf32>
   %c = affine.load %C[%i, %j] : memref<?x?xf32>
   %prod = arith.mulf %a, %b : f32
   %sum = arith.addf %c, %prod : f32
   affine.store %sum, %C[%i, %j] : memref<?x?xf32>
  }
 }
}
```

matmul_dhir.mlir

```
dhir.partition(%arg0, %arg1, %arg2, %agr3)
<{operandSegmentSizes = array<i32: 2, 1>}> ({
  affine.for %i = 0 to %arg0 {
   affine.for %j = 0 to %arg0 {
    affine.store %c0, %arg2[%i, %j] : memref<?x?xf32>
    affine.for %k = 0 to %arg0 {
     %2 = affine.load %arg0[%i, %k] : memref<?x?xf32>
     %3 = affine.load %arg1[%k, %j] : memref<?x?xf32>
     %4 = affine.load %arg2[%i, %j] : memref<?x?xf32>
     %5 = arith.mulf %2, %3 : f32
     %6 = arith.addf %4, %5 : f32
     affine.store %6, %arg2[%i, %j] : memref<?x?xf32>
 } } } dhir.return() })
```

LOWER-PARTITION CONVERSION

Transforms a Partition Op into multiple task operations according to the cluster config

PARTITION & COMMUNICATION ANALYSIS

- G=(AXB)X(CXD)
- Partition Analysis
- Broadcast Analysis

3mm_dhir_primitive.mlir

```
// Partition 1: arg7 NOT partitioned (accessed as
[k,j]), arg5 needBroadcast=true (used by partition 3)
target=node0, needBroadcast=true {
  dhir.task(%arg7, %subview_arg6[0:334],
%subview_arg5[0:334])
}
target=node1, needBroadcast=true {
  dhir.task(%arg7, %subview_arg6[334:667],
%subview_arg5[334:667])
}
target=node2, needBroadcast=true {
  dhir.task(%arg7, %subview_arg6[667:1000],
%subview_arg5[667:1000])
}
```

```
// Partition 2: arg10 NOT partitioned, arg8
needBroadcast=true (used by partition 3)
target=node0, needBroadcast=true {
  dhir.task(%subview_arg9[0:334], %arg10,
%subview_arg8[0:334])
}
target=node1, needBroadcast=true {
  dhir.task(%subview_arg9[334:667], %arg10,
%subview_arg8[334:667])
}
target=node2, needBroadcast=true {
  dhir.task(%subview_arg9[667:1000], %arg10,
%subview_arg8[667:1000])
}
```

```
// Partition 3: arg8 NOT partitioned (accessed as
[k,j])
target=node0, needBroadcast=false {
  dhir.task(%subview_arg5[0:334], %arg8,
%subview_arg11[0:334])
}
target=node1, needBroadcast=false {
  dhir.task(%subview_arg5[334:667], %arg8,
%subview_arg11[334:667])
}
target=node2, needBroadcast=false {
  dhir.task(%subview_arg5[667:1000], %arg8,
%subview_arg11[667:1000])
}
```

←

3mm_dhir_primitive.mlir

```
dhir.partition(%arg7, %arg6, %arg5)({
…
scf.for %arg12 = %c0 to %c1000 step %c1 {
 ...
 scf.for %arg13 = %c0 to %c1000 step %c1 {
 memref.store %cst, %arg5[%arg12, %arg13]
 ...
 scf.for %arg14 = %c0 to %c1000 step %c1 {
  %0 = memref.load %arg6[%arg12, %arg14]
  %1 = memref.load %arg7[%arg14, %arg13]
  %2 = arith.mulf %0, %1 : f32
  %3 = memref.load %arg5[%arg12, %arg13]
  %4 = arith.addf %3, %2 : f32
  memref.store %4, %arg5[%arg12, %arg13]
   }}}
 dhir.return() })
```

```
 dhir.partition(%arg9, %arg10, %arg8) ({
 ...
 scf.for %arg12 = %c0 to %c1000 step %c1 {
 ...
  scf.for %arg13 = %c0 to %c1000 step %c1 {
  memref.store %cst, %arg8[%arg12, %arg13]
  ...
  scf.for %arg14 = %c0 to %c1000 step %c1 {
   %0 = memref.load %arg9[%arg12, %arg14]
   %1 = memref.load %arg10[%arg14, %arg13]
   %2 = arith.mulf %0, %1 : f32
   %3 = memref.load %arg8[%arg12, %arg13]
   %4 = arith.addf %3, %2 : f32
   memref.store %4, %arg8[%arg12, %arg13]
   }}}
 dhir.return() })
```

```
 dhir.partition(%arg5, %arg8, %arg11)({
 ...
 scf.for %arg12 = %c0 to %c1000 step %c1 {
 ...
 scf.for %arg13 = %c0 to %c1000 step %c1 {
 memref.store %cst, %arg11[%arg12, %arg13]
 ...
 scf.for %arg14 = %c0 to %c1000 step %c1 {
  %0 = memref.load %arg5[%arg12, %arg14]
  %1 = memref.load %arg8[%arg14, %arg13]
  %2 = arith.mulf %0, %1
  %3 = memref.load %arg11[%arg12, %arg13]
  %4 = arith.addf %3, %2 : f32
  memref.store %4, %arg11[%arg12, %arg13]
   }}}
 dhir.return() })
```

LOWER-DHIR CONVERSION

Backend Specific DHIR Lowering with Inter and Intra Node Parallelism

SYNTHESIZING MPI PRIMITIVES

- Gather at Rank 0
- Broadcast to all ranks
- Barriers for synchronization

3mm_target.mlir

```
if (rank == 0) {
    omp.parallel {
      omp.loop_nest %i = 0 to 334 {
        scf.for %j ... scf.for %k {
          E[i,j] += A[i,k] * B[k,j]
    } } } }
if (rank == 1) { // ← GPU node
    gpu.memcpy  host → device
    gpu.launch_func @matmul_kernel(...)
    gpu.memcpy  device → host
}
if (rank == 2) { // ← GPU node
    gpu.memcpy  host → device
    gpu.launch_func @matmul_kernel(...)
    gpu.memcpy  device → host
}

// Gather E (needsBroadcast)
mpi.barrier()
if (rank == 0) {
    mpi.recv(E[334:667], from=node1)
    mpi.recv(E[667:1000],from=node2)
} else
    mpi.send(my_partition, to=node0)

//Broadcast full E to all nodes
  if (rank == 0) {
    for each node in [1..N]:
      mpi.send(E[0:1000], to=node)
  } else
     mpi.recv(E[0:1000], from=node0)

// GPU kernel (same for all partitions)
gpu.module @matmul_kernel {
  for %j ... for %k {
    C[i,j] += A[i,k] * B[k,j]
  }
}
```

←

3mm_dhir_primitive.mlir

```
// Partition 1: arg7 NOT partitioned (accessed as
[k,j]), arg5 needBroadcast=true (used by partition 3)
target=node0, needBroadcast=true {
  dhir.task(%arg7, %subview_arg6[0:334],
%subview_arg5[0:334])
}
target=node1, needBroadcast=true {
  dhir.task(%arg7, %subview_arg6[334:667],
%subview_arg5[334:667])
}
target=node2, needBroadcast=true {
  dhir.task(%arg7, %subview_arg6[667:1000],
%subview_arg5[667:1000])
}
```

```
// Partition 2: arg10 NOT partitioned, arg8
needBroadcast=true (used by partition 3)
target=node0, needBroadcast=true {
  dhir.task(%subview_arg9[0:334], %arg10,
%subview_arg8[0:334])
}
target=node1, needBroadcast=true {
  dhir.task(%subview_arg9[334:667], %arg10,
%subview_arg8[334:667])
}
target=node2, needBroadcast=true {
  dhir.task(%subview_arg9[667:1000], %arg10,
%subview_arg8[667:1000])
}
```

```
// Partition 3: arg8 NOT partitioned (accessed as
[k,j])
target=node0, needBroadcast=false {
  dhir.task(%subview_arg5[0:334], %arg8,
%subview_arg11[0:334])
}
target=node1, needBroadcast=false {
  dhir.task(%subview_arg5[334:667], %arg8,
%subview_arg11[334:667])
}
target=node2, needBroadcast=false {
  dhir.task(%subview_arg5[667:1000], %arg8,
%subview_arg11[667:1000])
}
```

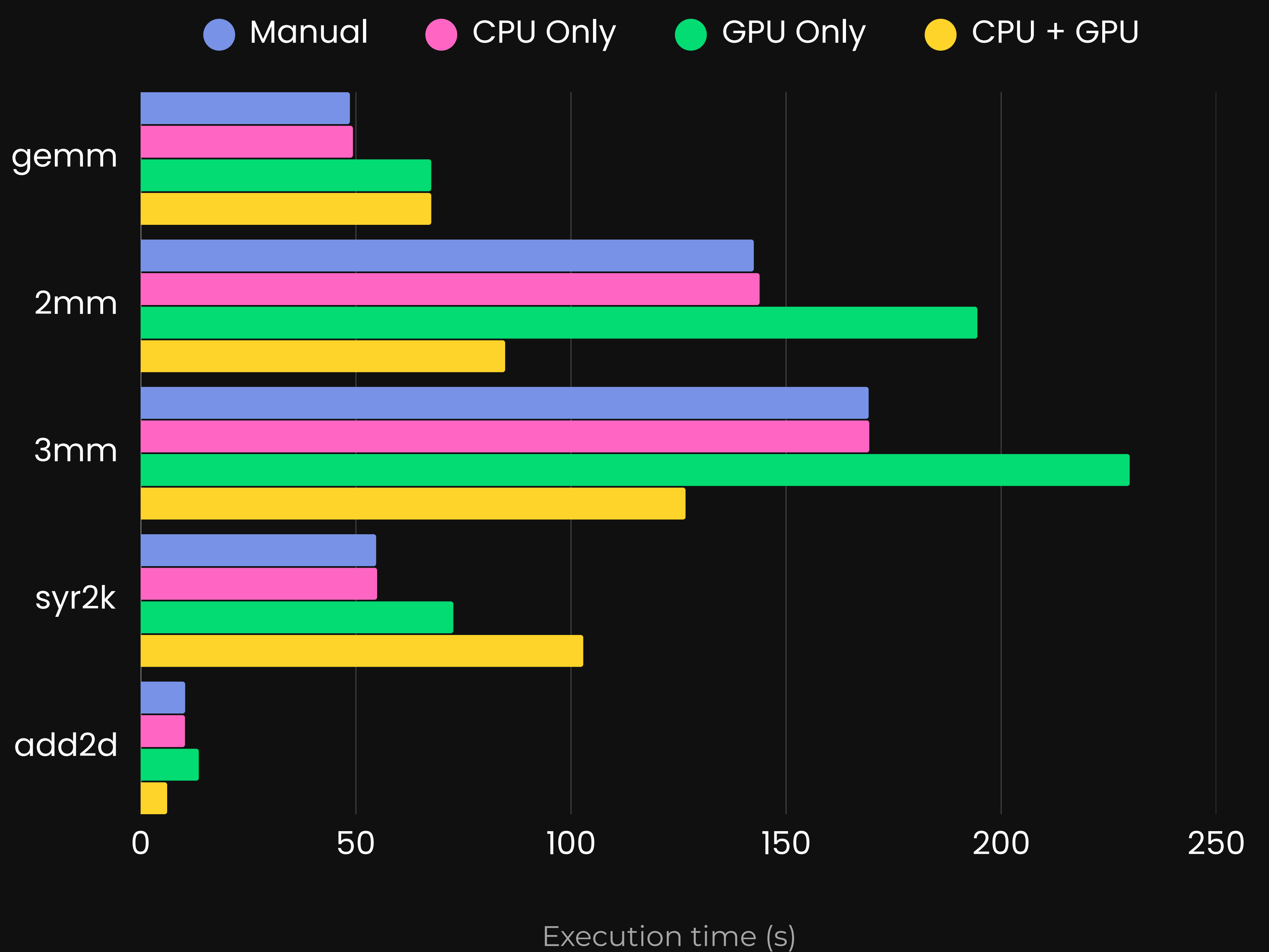
EXPERIMENTAL VALIDATION

Using ***PolyBench*** kernels on Extra Large datasets (10k x 10k matrices).

CONFIGURATIONS

- CPU Only
  3 Nodes, 96 Cores Total
- GPU Only
  3 Nodes, 3 GPUs
- Heterogeneous
  3 Nodes, 96 Cores, 3 GPUs

0 Lines of Code Changed



CONCLUSION

DHIR bridges the gap between programmability and the complex reality of modern HPC clusters

RELEVANCE OF DHIR

- Growth of AI Accelerator Companies
- MLIR as the Industry Standard Compiler Infrastructure
- The Sweet Spot of DHIR

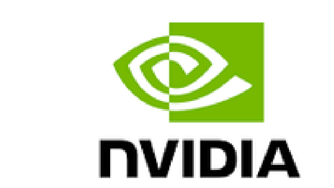
FUTURE

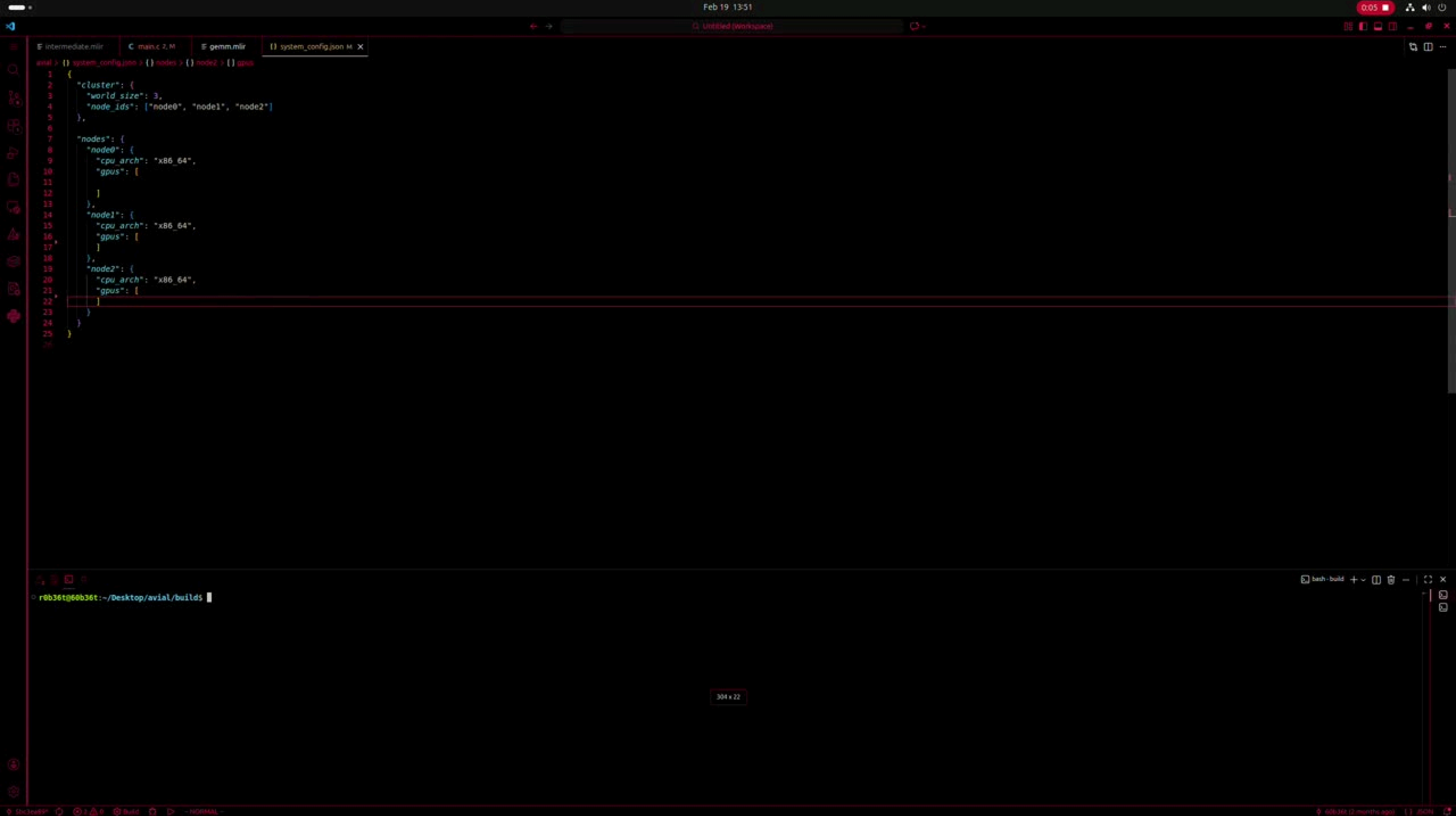
- Advanced Cost Model
- Communication Optimization

This work was presented at **PLDI SRC 2025** and has been submitted to **ICS 2026**.

EXPLORE THE PROJECT

Scan the QR code or visit *github.com/johnmaxrin/avial* to check out the project.




Feb 19 13:51
Untitled (Workspace)
intermediate.mlir
main.c
gemm.mlir
system_config.json
{
"cluster": {
"world_size": 3,
"node_ids": ["node0", "node1", "node2"]
},
"nodes": {
"node0": {
"cpu_arch": "x86_64",
"gpus": [
]
},
"node1": {
"cpu_arch": "x86_64",
"gpus": [
]
},
"node2": {
"cpu_arch": "x86_64",
"gpus": [
]
}
}
}
r0b36t@60b36t:~/Desktop/avial/build$
bash - build
304 x 22
NORMAL