An MLIR Dialect for **Distributed Heterogeneous** Computing

Presenter
ROBERT K SAMUEL

Advisor
RUPESH NASRE

Role of **Dialects** in MLIR

- **Extensible** Compiler Infrastructure
- Encapsulates a **Domain-Specific abstraction**
- Structured transformations and **Progressive lowering**
- Great for targeting **individual** hardware, But 🤔

[1] Infrastructure to define IRs and transformations.


```
C = A + B
```

↓

```
Assignment
├── LHS: C
└── RHS: BinaryOp(+)
    ├── Operand: A
    └── Operand: B
```

→

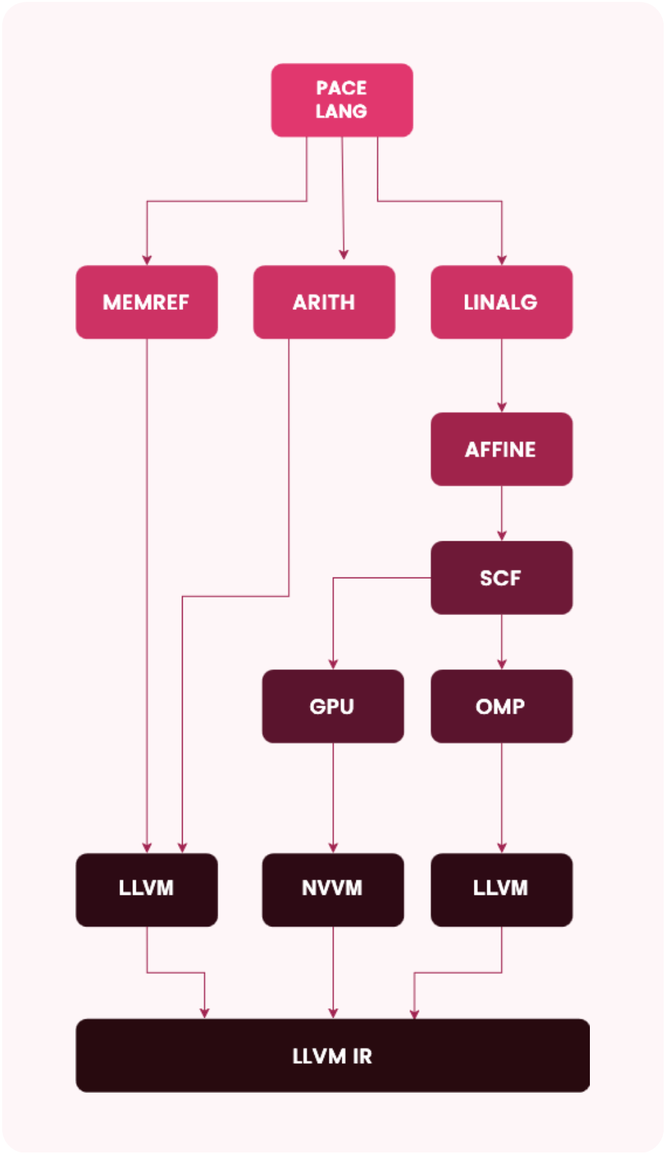
```
%a = memref.load %A
%b = memref.load %B
%c = arith.add %a, %b
memref.store %C, %c
```

↓

```
%a = load i32, i32* %A
%b = load i32, i32* %B
%c = add i32 %a, %b
store int %c, i32* %C
```

→

```
$ clang++ dsl.ll
$ ./a.out
```



MPI

Why is **Distributed Heterogeneous** Computing Hard?

Architectural Chaos

CPUs, GPUs, FPGAs each follow different memory models, execution semantics, and APIs.

Manual Coordination & Fragmented Tooling

Developers juggle low-level data movement and multiple tool-chains, making the workflow error-prone and hard to debug.

Missed Optimization Potential

Without a global view, compilers can't fuse tasks, optimize placement, or utilize hardware efficiently.

```
scf.parallel %i = 0 to 20
{
   %x = memref.load %a[%i]
   %y = memref.load %b[%i]
   %z = arith.addi %x, %y
   memref.store %z, %c[%i]
}
```

OpenMP

AMD ROCm

The Missing Piece in MLIR

MLIR abstracts heterogeneous hardware, but lacks support for **distributed orchestration**. A dialect for cluster-wide task execution fills this critical gap.

Unifying Devices, Nodes, and Execution

Problem	Solution
Architectural Chaos	Explicit **TargetOp** for device-level scheduling
Manual Coordination & Fragmented Tooling	Unified **ScheduleOp** for task orchestration
Missed Optimization Potential	Task dependencies + metadata → smart lowering
Cluster-wide Orchestration	First-class **TaskOp** model for distributed systems

Drop-In Approach

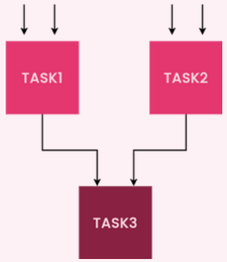
Developers define compute using **TaskOps**, While the compiler automatically manages placement, scheduling, and execution across CPUs, GPUs, and distributed clusters.

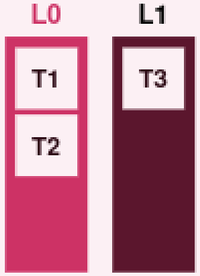
```
avial.schedule %a, %b, %c, %d
{
 %cpu = avial.target{arch = "x86_64"}
 %gpu = avial.target{arch = “sm_90” }
 %task1 = avial.task %cpu, %a, %b, %c
 {
    scf.parallel %i = 0 to 20
    {
      %x = memref.load %a[%i]
      %y = memref.load %b[%i]
      %z = arith.addi %x, %y
      memref.store %z, %c[%i]
    }
 }

 %task2 = avial.task %cpu, %a, %b, %d
 {
    scf.parallel %i = 0 to 20
    {
      ...
      memref.store %z, %c[%i]
    }
  }

 %task3 = avial.task %gpu, %c, %d, %d
 {
    gpu.launch
    {
      %tid = gpu.thread_id x
      ...
      memref.store %z, %c[%tid]
    }
  }

}
```

Scheduling for Distributed Heterogeneous Environments

```
avial.target : $targetAttrs → $targetRef

avial.task : $targetRef, $reads, $writes
{

  // task region

} → $taskRef


avial.schedule : $memrefs
{
  // Memrefs
  // Task refs
}
```

- Construct Dependency **DAG**
- Perform Topological Sort
- Parallel Execution per Level
- Insert Synchronization Barriers

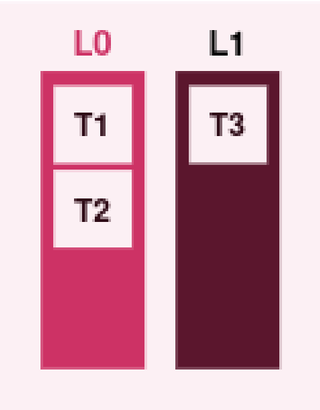




Existing Pipleline | **AVIAL** | **MPI** | **LLVM IR** | **BIN**

```
avial.schedule %a, %b, %c, %d
{
 %cpu = avial.target{arch = "x86_64"}
 %gpu = avial.target{arch = “sm_90” }
 %task1 = avial.task %cpu, %a, %b, %c
 {
     scf.parallel %i = 0 to 20
     {
       %x = memref.load %a[%i]
       %y = memref.load %b[%i]
       %z = arith.addi %x, %y
       memref.store %z, %c[%i]
     }
 }

 %task2 = avial.task %cpu, %a, %b, %d
 {
     scf.parallel %i = 0 to 20
     {
       ...
       memref.store %z, %c[%i]
     }
  }

 %task3 = avial.task %gpu, %c, %d, %d
 {
     gpu.launch
     {
       %tid = gpu.thread_id x
       ...
       memref.store %z, %c[%tid]
     }
  }

}
```

Lowering to MPI




```
if(rank == 0)
{
  // Do this
}

else if(rank == 1)
{
  // Do Something Else
}
```

Typical MPI Program

```
func @genmpi(%a, %b, %c, %d) {
  %comm = mpi.comm_world
  %rnk = mpi.comm_rank
  %sz = mpi.comm_size

  %p0 = arith.remsi 0, %sz
  %cnd1 = arith.cmpi eq, %rnk, %p0
    scf.if %cnd1 {
     scf.parallel %i = 0 to 20
     {
       %x = memref.load %a[%i]
       %y = memref.load %b[%i]
       %z = arith.addi %x, %y
       memref.store %z, %c[%i]
     }
    }

  %p1 = arith.remsi 1, %sz
  %cnd2 = arith.cmpi eq, %rnk, %p1
    scf.if %cnd2 {
      loop
      mpi.send %d 0
    }
  mpi.barrier(%comm)

  %p3 = arith.remsi 0, %sz
  %cnd3 = arith.cmpi eq, %rnk, %p3
   scf.if %cnd3 {
    mpi.recv %d 1
    gpu.launch {
      loop
      gpu.terminator
    }
   }

  }
}
```

```
avial.schedule %a, %b, %c, %d
{
 %cpu = avial.target{arch = "x86_64"}
 %gpu = avial.target{arch = “sm_90” }
 %task1 = avial.task %cpu, %a, %b, %c
 {
     scf.parallel %i = 0 to 20
     {
       %x = memref.load %a[%i]
       %y = memref.load %b[%i]
       %z = arith.addi %x, %y
       memref.store %z, %c[%i]
     }
 }

 %task2 = avial.task %cpu, %a, %b, %d
 {
     scf.parallel %i = 0 to 20
     {
       ...
       memref.store %z, %c[%i]
     }
  }

 %task3 = avial.task %gpu, %c, %d, %d
 {
     gpu.launch
     {
       %tid = gpu.thread_id x
       ...
       memref.store %z, %c[%tid]
     }
  }

}
```

END