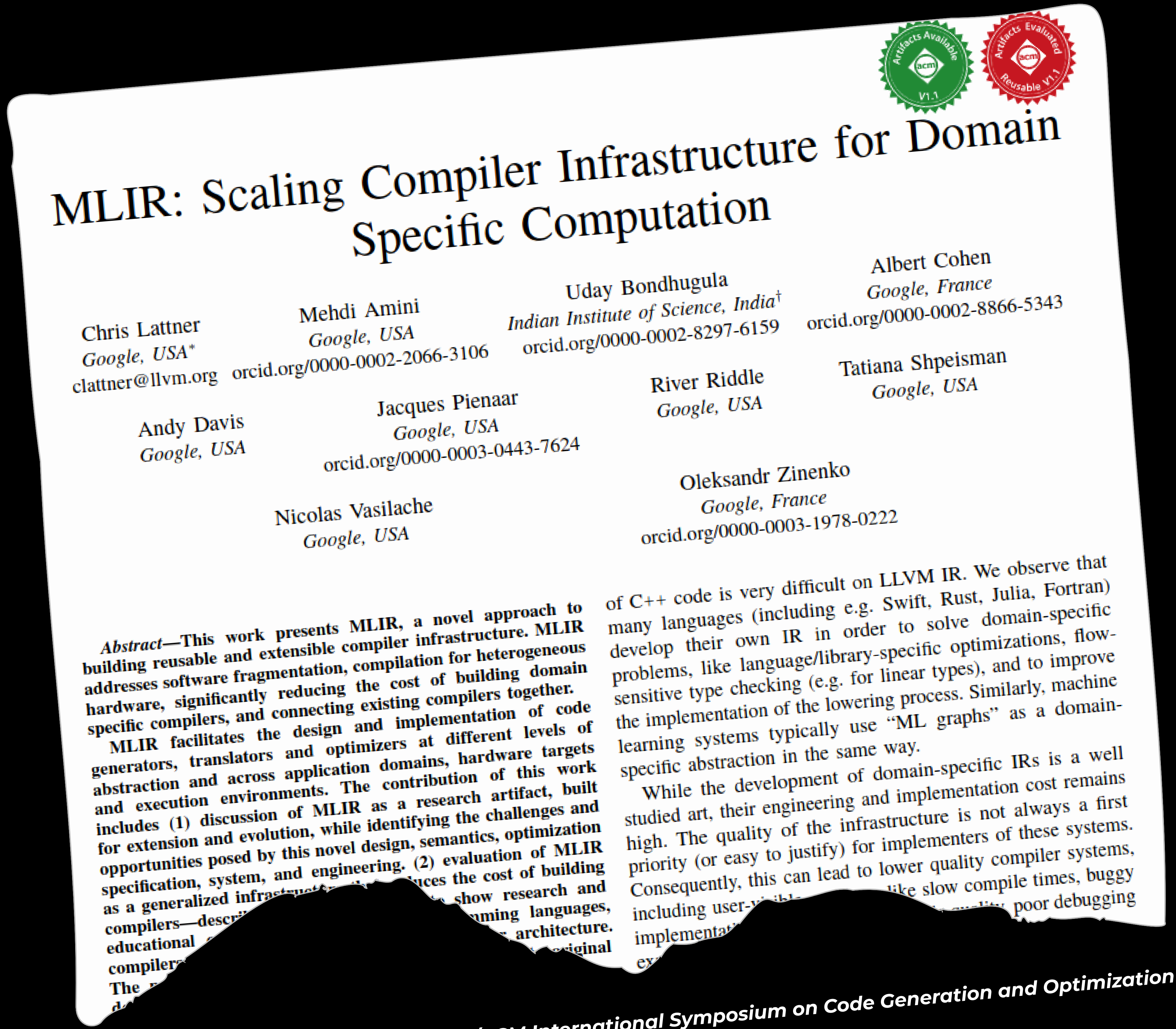
MLIR Primer

Multi-dimensional **L**oop **IR**

Machine **L**earning **IR**

Mid-**L**evel **IR**

Multi-**L**evel **IR**



MLIR: Scaling Compiler Infrastructure for Domain Specific Computation

Chris Lattner
Google, USA*
clattner@llvm.org

Mehdi Amini
Google, USA
orcid.org/0000-0002-2066-3106

Uday Bondhugula
Indian Institute of Science, India†
orcid.org/0000-0002-8297-6159

Albert Cohen
Google, France
orcid.org/0000-0002-8866-5343

Andy Davis
Google, USA

Jacques Pienaar
Google, USA
orcid.org/0000-0003-0443-7624

River Riddle
Google, USA

Tatiana Shpeisman
Google, USA

Nicolas Vasilache
Google, USA

Oleksandr Zinenko
Google, France
orcid.org/0000-0003-1978-0222

Abstract—This work presents MLIR, a novel approach to building reusable and extensible compiler infrastructure. MLIR addresses software fragmentation, compilation for heterogeneous hardware, significantly reducing the cost of building domain specific compilers, and connecting existing compilers together. MLIR facilitates the design and implementation of code generators, translators and optimizers at different levels of abstraction and across application domains, hardware targets and execution environments. The contribution of this work includes (1) discussion of MLIR as a research artifact, built for extension and evolution, while identifying the challenges and opportunities posed by this novel design, semantics, optimization specification, system, and engineering. (2) evaluation of MLIR as a generalized infra... ...uces the cost of building compilers—descri... ...show research and educational ... ...mming languages, compilers... ...architecture. The ... ...riginal

of C++ code is very difficult on LLVM IR. We observe that many languages (including e.g. Swift, Rust, Julia, Fortran) develop their own IR in order to solve domain-specific problems, like language/library-specific optimizations, flow-sensitive type checking (e.g. for linear types), and to improve the implementation of the lowering process. Similarly, machine learning systems typically use "ML graphs" as a domain-specific abstraction in the same way.

While the development of domain-specific IRs is a well studied art, their engineering and implementation cost remains high. The quality of the infrastructure is not always a first priority (or easy to justify) for implementers of these systems. Consequently, this can lead to lower quality compiler systems, including user-vi... ...like slow compile times, buggy implementati... ...poor debugging ex...

*CGO '21: Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization

C/C++
PyTorch
.
.
StarPlat

LLVM IR

LLVM Infra

Binary

x86_64
arm
.
.
RISCV

InstCombine
SimplifyCFG
Reassociate
GVN
SCCP
CorrelatedValuePropagation
DeadCodeElimination (DCE)
DeadStoreElimination (DSE)
AggressiveDCE
ConstantPropagation
ConstantHoisting
EarlyCSE
RedundantDbgInstElimination
JumpThreading
TailDuplication
IndVarSimplify
StrengthReduce
Sinking
CodeGenPrepare
DemoteRegisterToMemory
SimplifyCFG
MergedLoadStoreMotion
LowerExpectIntrinsic

LoopSimplify
LoopRotate
LoopUnswitch
LoopUnroll
LoopUnrollAndJam
LoopDeletion
LoopInstSimplify
LICM
LoopIdiomRecognize
LoopDistribute
LoopFuse
LoopInterchange
LoopLoadElim
LoopPredication
LoopReroll
LoopSimplifyCFG
LoopStrengthReduce
LoopVectorize
LoopSink
LoopFlatten
SampleProfileLoader
ThinLTO
LTO

Mem2Reg
SROA
MergedLoadStoreMotion
GVNHoist
GVNHoistSink
MemCpyOpt
GlobalValueNumbering
AlignmentFromAssumptions
DSE (DeadStoreElimination)
ADCE (AggressiveDCE)
LICM
LoadStoreVectorizer
MergeICmps
PartiallyInlineLibCalls
Scalarizer
CrossDSOCFI
MergeFunctions
CalledValuePropagation
EliminateAvailableExternally
InlineCostAnalysis
HotColdSplitting
PartialInlining
LowerTypeTests

Inliner
AlwaysInliner
GlobalOptimizer
GlobalDCE
FunctionAttrs
ArgumentPromotion
IPSCCP
ConstantMerge
DeadArgumentElimination
PruneEH
WholeProgramDevirt
LoopVectorize
SLPVectorizer
VectorCombine
InterleavedAccessPass
Polly
InstrProfiling
ObjCARCOpts
BDCE
DbgValueHistoryCalculator
BarrierNoopPass
Verifier
BlockPlacement

InstructionSelect
MachineLICM
MachineCSE
MachineGVN
MachineCopyPropagation
MachineCombiner
MachineScheduler
RegisterCoalescer
PrologEpilogInserter
BranchFolding
TailDuplication
MachineBlockPlacement
CodePlacementOpt
MachineDominatorTree
MachineBranchProbabilityInfo
MachineBlockFrequencyInfo
PGOInstrumentationGen
PGOInstrumentationUse
SampleProfile
SamplePGO
MachineOutliner
FunctionReordering
CallSiteSplitting

So, what's missing?

We still face several fundamental issues when building modern compilers.

```
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;
}
```

```
...
%69 = mul i64 %52, 128
%70 = add i64 %69, %65
%71 = getelementptr float, ptr %68, i64 %70
%72 = load float, ptr %71, align 4
%73 = extractvalue { ptr, ptr, i64, [2 x i64], [2 x i64] } %38, 1
%74 = mul i64 %65, 128
%75 = add i64 %74, %57
%76 = getelementptr float, ptr %73, i64 %75
%77 = load float, ptr %76, align 4
%78 = fmul float %72, %77
%79 = extractvalue { ptr, ptr, i64, [2 x i64], [2 x i64] } %31, 1
%80 = mul i64 %52, 128
%81 = add i64 %80, %57
%82 = getelementptr float, ptr %79, i64 %81
%83 = load float, ptr %82, align 4
%84 = fadd float %83, %78
%85 = extractvalue { ptr, ptr, i64, [2 x i64], [2 x i64] } %31, 1
%86 = mul i64 %52, 128
%87 = add i64 %86, %57
%88 = getelementptr float, ptr %85, i64 %87
store float %84, ptr %88, align 4
%89 = add i64 %65, 1
...
```

Vertex2Edge? Auto-Parallelization? Push2Pull?

StarPlat IR

```
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;
}
```

```
module {
  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
  }
}
```

```
...
%69 = mul i64 %52, 128
%70 = add i64 %69, %65
%71 = getelementptr float, ptr %68, i64 %70
%72 = load float, ptr %71, align 4
%73 = extractvalue { ptr, ptr, i64, [2 x i64], [2 x i64] } %38, 1
%74 = mul i64 %65, 128
%75 = add i64 %74, %57
%76 = getelementptr float, ptr %73, i64 %75
%77 = load float, ptr %76, align 4
%78 = fmul float %72, %77
%79 = extractvalue { ptr, ptr, i64, [2 x i64], [2 x i64] } %31, 1
%80 = mul i64 %52, 128
%81 = add i64 %80, %57
%82 = getelementptr float, ptr %79, i64 %81
%83 = load float, ptr %82, align 4
%84 = fadd float %83, %78
%85 = extractvalue { ptr, ptr, i64, [2 x i64], [2 x i64] } %31, 1
%86 = mul i64 %52, 128
%87 = add i64 %86, %57
%88 = getelementptr float, ptr %85, i64 %87
store float %84, ptr %88, align 4
%89 = add i64 %65, 1
...
```

MLIR provides infrastructure to do this!

Who Uses MLIR?

Design Principles

Little Builtin, Everything Customizable

Minimal Core Concepts

Full Customizability

Reusable Infrastructure

Broad Expressiveness

Ecosystem Discipline

; Keep only the essentials built-in,
everything else be custom.

```
module {
  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
  }
}
```

- Attributes
  - AffineMapAttr
  - ArrayAttr
  - DenseArrayAttr
  - DenseIntOrFPElementsAttr
  - DenseResourceElementsAttr
  - DenseStringElementsAttr
  - DictionaryAttr
  - FloatAttr
  - IntegerAttr
  - IntegerSetAttr
  - OpaqueAttr
  - SparseElementsAttr
  - StringAttr
  - SymbolRefAttr
  - TypeAttr
  - UnitAttr
  - StridedLayoutAttr
- Location Attributes
  - CallSiteLoc
  - FileLineColRange
  - FusedLoc
  - NameLoc
  - OpaqueLoc
  - UnknownLoc
- DistinctAttribute
- Operations
  - builtin.module (ModuleOp)
  - builtin.unrealized_conversion_cast (UnrealizedConversionCastOp)
- Types
  - BFloat16Type
  - ComplexType
  - Float4E2M1FNType
  - Float6E2M3FNType
  - Float6E3M2FNType
  - Float8E3M4Type
  - Float8E4M3Type
  - Float8E4M3B11FNUZType
  - Float8E4M3FNType
  - Float8E4M3FNUZType
  - Float8E5M2Type
  - Float8E5M2FNUZType
  - Float8E8M0FNUType
  - Float16Type
  - Float32Type
  - Float64Type
  - Float80Type
  - Float128Type
  - FloatTF32Type
  - FunctionType
  - GraphType
  - IndexType
  - IntegerType
  - MemRefType
  - NoneType
  - OpaqueType
  - RankedTensorType
  - TupleType
  - UnrankedMemRefType
  - UnrankedTensorType
  - VectorType
- Type Interfaces

Progressive lowering

Multi-Level Abstraction

Step-by-Step Lowering

Extensible Design

Unified Transformation

Pass Interaction

; Enables flexible, multi-level lowering with small, composable steps for optimized performance across abstraction levels.

```
module {
  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
  }
}
```

```
module {
  func.func @Compute_TC(%g : !starplat.graph) -> i64 {
    %0 = arith.constant 0
    %triangle_count = llvm.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 = llvm.load %triangle_count
            %one = arith.constant 1
            %new = arith.add %old, %one
            llvm.store %new, %triangle_count
          }
        }
      }
    }
    %ret = llvm.load %triangle_count : i64
    func.return %ret
  }
}
```

Maintain higher-level semantics

Preserve High-Level Abs

Avoid “Raising” Semantics

Conscious Lowering

Mixed Abstraction Levels

Heterogeneous and Parallel

; MLIR preserves semantic and structural information for improved analysis, parallelization, and heterogeneous code generation, lowering it only when necessary.

```
module {
  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
  }
}
```

```
module {
  func.func @Compute_TC(%g : !starplat.graph) -> i64 {
    %0 = arith.constant 0
    %triangle_count = memref.alloca i64

    omp.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
  }
}
```

IR Design Details

Operations

In MLIR, everything is an operation. Not just instructions, but the entire hierarchy of computation.

Dialect *Operands*

```
%1 = arith.addi %a, %b : i32
```

Result *Op* *Return Type*

Component	Description
Opcode	Unique string: <dialect>.<operation> (e.g., arith.addi)
Operands	Input SSA values consumed by the op
Results	Output SSA values produced by the op
Attributes	Immutable metadata (e.g., constants, names, options)
Regions / Blocks	Contain nested operations — enable control flow
Location Info	Source position (for debug, diagnostics)

```
%results:2 = "d.operation"(%arg0, %arg1) ({
 // Regions belong to Ops and can have multiple blocks.
 ^block(%argument: !d.type):
  // Ops have function types (expressing mapping).
  %value = "nested.operation"() ({
   // Ops can contain nested regions.
   "d.op"() : () -> ()
  }) : () -> (!d.other_type)
  "consume.value"(%value) : (!d.other_type) -> ()

 ^other_block:
  "d.terminator"() [^block(%argument : !d.type)] : () -> ()
 })
// Ops can have a list of attributes.
{attribute="value" : !d.type} : () -> (!d.type, !d.other_type)
```

No fixed set of ops, fully extensible system

Traits & interfaces let passes reason about semantics

Dialects

Dialects are namespaces that logically group Ops, Types, and Attributes together.

They make MLIR modular, extensible, and organized, like compiler libraries for different domains.

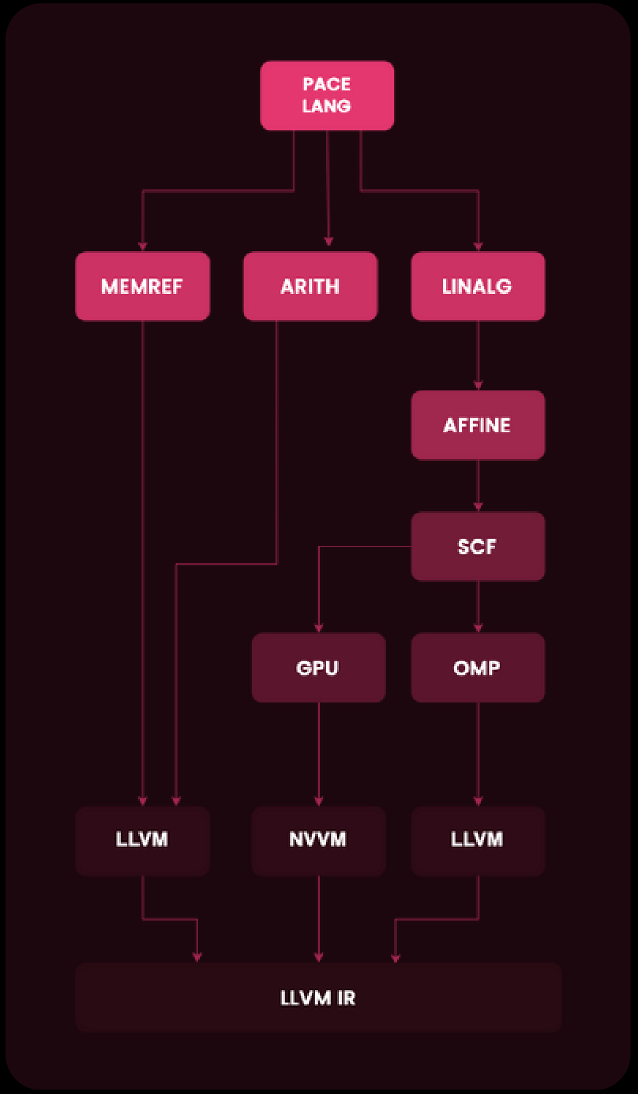
Enable modular compiler design

Let developers define domain-specific abstractions (e.g., Tensor, GPU, Quantum)

Allow progressive lowering

Prevent naming collisions and keep IR organized

Dialects make MLIR truly extensible. You can build your own world of operations and still interoperate with everyone else's.



'acc' Dialect
'affine' Dialect
'amdgpu' Dialect
'amx' Dialect
'arith' Dialect
'arm_neon' Dialect
'arm_sve' Dialect
'ArmSME' Dialect
'async' Dialect
'bufferization' Dialect
'cf' Dialect
'complex' Dialect
'dlti' Dialect
'emitc' Dialect
'func' Dialect
'gpu' Dialect
'index' Dialect
'irdl' Dialect
'linalg' Dialect
'llvm' Dialect
'math' Dialect
'pdl' Dialect
'ptr' Dialect
'quant' Dialect
'rocdl' Dialect
'smt' Dialect
'sparse_tensor' Dialect
'tensor' Dialect
'ub' Dialect
'vcix' Dialect
'vector' Dialect
'wasmssa' Dialect
'x86vector' Dialect
'xegpu' Dialect
'xevm' Dialect
Builtin Dialect
OpInterface definitions
SPIR-V Dialect
TOSA Dialect
Transform Dialect
'memref' Dialect
'ml_program' Dialect
'mpi' Dialect
'nvgpu' Dialect
'nvvm' Dialect
'omp' Dialect
'pdl_interp' Dialect
'scf' Dialect
'shape' Dialect
'shard' Dialect

Why MLIR Needs Debugging?

```
module {
 func.func @matmul(%arg0: i32, %arg1: i32, %arg2: i32, %arg3: 
memref<?x128xf32>, %arg4: memref<?x128xf32>, %arg5: memref<?
x128xf32>) {
   %cst = arith.constant 0.000000e+00 : f32
   %0 = arith.index_cast %arg0 : i32 to index
   %1 = arith.index_cast %arg1 : i32 to index
   %2 = arith.index_cast %arg2 : i32 to index
   affine.for %arg6 = 0 to 128 {
    affine.for %arg7 = 0 to 128 {
    affine.store %cst, %arg5[%arg6, %arg7] : memref<?x128xf32>
    affine.for %arg8 = 0 to 128 {
    %3 = affine.load %arg3[%arg6, %arg8] : memref<?x128xf32>
    %4 = affine.load %arg4[%arg8, %arg7] : memref<?x128xf32>
    %5 = arith.mulf %3, %4 : f32
    %6 = affine.load %arg5[%arg6, %arg7] : memref<?x128xf32>
    %7 = arith.addf %6, %5 : f32
    affine.store %7, %arg5[%arg6, %arg7] : memref<?x128xf32>
    }
   }
  }
  return
 }
}
```

What value does %arg0 actually hold at runtime?

Is my loop even running?

Am I accidentally accessing memory out of bounds?

```
int main()
{
    float A[10], B[10], C[10];
    int N=10;
   for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      for (int k = 0; k < N; ++k)
        C[i*N + j] += A[i*N + k] * B[k*N + j];
    return 0;
}
```

What you would do in this scenario?

```
module {
 func.func @matmul(%arg0: i32, %arg1: i32, %arg2: i32, %arg3: 
memref<?x128xf32>, %arg4: memref<?x128xf32>, %arg5: memref<?
x128xf32>) {
   arey.print %arg0 : i32
   %cst = arith.constant 0.000000e+00 : f32
   %0 = arith.index_cast %arg0 : i32 to index
   %1 = arith.index_cast %arg1 : i32 to index
   %2 = arith.index_cast %arg2 : i32 to index
   affine.for %arg6 = 0 to 128 {
   arey.print_str “Hi Here”
    affine.for %arg7 = 0 to 128 {
    affine.store %cst, %arg5[%arg6, %arg7] : memref<?x128xf32>
    affine.for %arg8 = 0 to 128 {
    %3 = affine.load %arg3[%arg6, %arg8] : memref<?x128xf32>
    %4 = affine.load %arg4[%arg8, %arg7] : memref<?x128xf32>
    arey.assert %arg7 : i32 eq 1
    ...
    }
   }
  }
  return
 }
}
```

Aspect	MLIR	LLVM
Domain Specific	Designed to support domain-specific dialects (e.g., Tensor, GPU, Linalg).	Not domain-specific; single fixed IR for all domains.
Custom Operation	Users can define custom operations via dialects.	Users can’t define custom operations via dialects.
Abstraction Level	Multi-level	Low-level only
Infrastructure Support	✓	✓

Multi-dimensional **L**oop **IR**?

Machine **L**earning **IR**?

Mid-**L**evel **IR**?

Multi-**L**evel **IR**?

What is MLIR?

Ongoing works @Gajendra

StarPlat MLIR

MLIR for graph analytics

Operations / Types related to Graph Analytivs

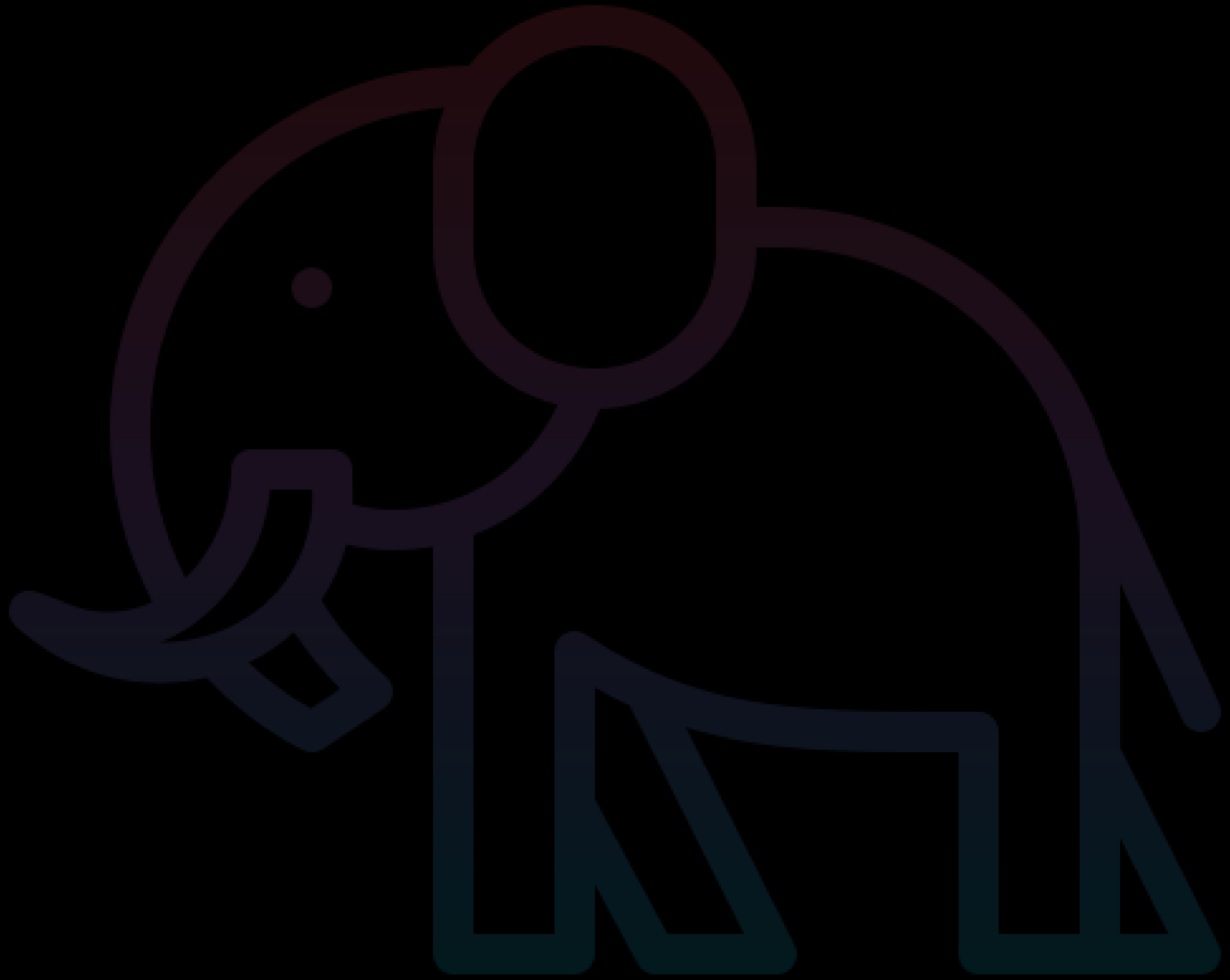
Vertex2Edge , Edge2Vertex, etc

DHIR

Distributed Heterogeneous IR

Orchestrates tasks

Auto parallelization, Task Scheduling, etc



Q&A