Dialects in LLVM IR

An Example using Cooperative Matrices

Cooperative Matrices

- AI applications require lots of matrix-matrix multiplications
- New Vulkan shader extension introduces the cooperative matrix type
- Special data layout depending on hardware support
- Some requirements on AMD GPUs:
  - When calculating $A \cdot B + C$, A needs to be transposed

Transpose in LLVM IR

```
958 define <8 x float> @test_transpose_simple(<8 x float> %a) {
959   %1 = call i32 @llvm.amdgcn.mbcnt.lo(i32 -1, i32 0)
960   %2 = call i32 @llvm.amdgcn.mbcnt.hi(i32 -1, i32 %1)
961   %3 = and i32 %2, 1
962   %4 = icmp eq i32 %3, 0
963   %5 = extractelement <8 x float> %a, i64 0
964   %6 = bitcast float %5 to i32
965   %7 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %6, i32 14570689)
966   %8 = bitcast i32 %7 to float
967   %9 = extractelement <8 x float> %a, i64 1
968   %10 = bitcast float %9 to i32
969   %11 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %10, i32 14570689)
970   %12 = bitcast i32 %11 to float
971   %13 = extractelement <8 x float> %a, i64 2
972   %14 = bitcast float %13 to i32
973   %15 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %14, i32 14570689)
974   %16 = bitcast i32 %15 to float
975   %17 = extractelement <8 x float> %a, i64 3
976   %18 = bitcast float %17 to i32
977   %19 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %18, i32 14570689)
978   %20 = bitcast i32 %19 to float
979   %21 = extractelement <8 x float> %a, i64 4
980   %22 = bitcast float %21 to i32
981   %23 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %22, i32 14570689)
982   %24 = bitcast i32 %23 to float
983   %25 = extractelement <8 x float> %a, i64 5
984   %26 = bitcast float %25 to i32
985   %27 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %26, i32 14570689)
986   %28 = bitcast i32 %27 to float
987   %29 = extractelement <8 x float> %a, i64 6
988   %30 = bitcast float %29 to i32
989   %31 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %30, i32 14570689)
990   %32 = bitcast i32 %31 to float
991   %33 = extractelement <8 x float> %a, i64 7
992   %34 = bitcast float %33 to i32
993   %35 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %34, i32 14570689)
```

Transpose as high-level operation

```
4 define <8 x float> @test_transpose_simple(<8 x float> %a) {
5   %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6   ret <8 x float> %a.t
7 }
```

```
958 define <8 x float> @test_transpose_simple(<8 x float> %a) {
959   %1 = call i32 @llvm.amdgcn.mbcnt.lo(i32 -1, i32 0)
960   %2 = call i32 @llvm.amdgcn.mbcnt.hi(i32 -1, i32 %1)
961   %3 = and i32 %2, 1
962   %4 = icmp eq i32 %3, 0
963   %5 = extractelement <8 x float> %a, i64 0
964   %6 = bitcast float %5 to i32
965   %7 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %6, i32 14570689)
966   %8 = bitcast i32 %7 to float
967   %9 = extractelement <8 x float> %a, i64 1
968   %10 = bitcast float %9 to i32
969   %11 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %10, i32 14570689)
970   %12 = bitcast i32 %11 to float
971   %13 = extractelement <8 x float> %a, i64 2
972   %14 = bitcast float %13 to i32
973   %15 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %14, i32 14570689)
974   %16 = bitcast i32 %15 to float
975   %17 = extractelement <8 x float> %a, i64 3
976   %18 = bitcast float %17 to i32
977   %19 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %18, i32 14570689)
978   %20 = bitcast i32 %19 to float
979   %21 = extractelement <8 x float> %a, i64 4
980   %22 = bitcast float %21 to i32
981   %23 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %22, i32 14570689)
982   %24 = bitcast i32 %23 to float
983   %25 = extractelement <8 x float> %a, i64 5
984   %26 = bitcast float %25 to i32
985   %27 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %26, i32 14570689)
986   %28 = bitcast i32 %27 to float
987   %29 = extractelement <8 x float> %a, i64 6
988   %30 = bitcast float %29 to i32
989   %31 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %30, i32 14570689)
990   %32 = bitcast i32 %31 to float
991   %33 = extractelement <8 x float> %a, i64 7
992   %34 = bitcast float %33 to i32
993   %35 = call i32 @llvm.amdgcn.mov.dpp8.i32(i32 %34, i32 14570689)
```

Optimizing Operations

```
define <8 x float> @test_load_transpose(ptr addrspace(3) %ptr) {
  %a = call <8 x float> @lgc.cooperative.matrix.load.v8f32(ptr addrspace(3) %ptr, i32 4, i1 true, i32 1, i32 0, i32 0)
  %t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
  ret <8 x float> %t
}
```

```
define <8 x float> @test_load_transpose(ptr addrspace(3) %ptr) {
  %a = call <8 x float> @lgc.cooperative.matrix.load.v8f32(ptr addrspace(3) %ptr, i32 4, i1 false, i32 1, i32 0, i32 0)
  ret <8 x float> %a
}
```

Working with new intrinsics – current approach

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

Working with new intrinsics – current approach

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
const static char CooperativeMatrixTranspose[] = "lgc.cooperative.matrix.transpose";
```

Working with new intrinsics – current approach

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
const static char CooperativeMatrixTranspose[] = "lgc.cooperative.matrix.transpose";
```

```
std::string callName(s: lgcName::CooperativeMatrixTranspose);
Value *args[] = {[0]=matrix, [1]=elementType, [2]=layout};

CallInst *result = CreateNamedCall(funcName: callName, retTy: matrix->getType(),
                                   args, attribs: {[0]=Attribute::ReadOnly, [1]=Attribute::WillReturn});
```

Working with new intrinsics – current approach

```
const static char CooperativeMatrixTranspose[] = "lgc.cooperative.matrix.transpose";
```

```
std::string callName(s: lgcName::CooperativeMatrixTranspose);
Value *args[] = {[0]=matrix, [1]=elementType, [2]=layout};

CallInst *result = CreateNamedCall(funcName: callName, retTy: matrix->getType(),
                                   args, attribs: {[0]=Attribute::ReadOnly, [1]=Attribute::WillReturn});
```

```
if (auto *call: CallInst * = dyn_cast<CallInst>(Val: input)) {
  if (auto *callee: Function * = call->getCalledFunction()) {
    if (callee->getName().starts_with(Prefix: lgcName::CooperativeMatrixTranspose)) {
      Value *src = call->getArgOperand(i: 0);
```

Working with new intrinsics – current approach

```
const static char CooperativeMatrixTranspose[] = "lgc.cooperative.matrix.transpose";
```

```
std::string callName(s: lgcName::CooperativeMatrixTranspose);
Value *args[] = {[0]=matrix, [1]=elementType, [2]=layout};

CallInst *result = CreateNamedCall(funcName: callName, retTy: matrix->getType(),
                                   args, attribs: {[0]=Attribute::ReadOnly, [1]=Attribute::WillReturn});
```

```
if (auto *call: CallInst * = dyn_cast<CallInst>(Val: input)) {
  if (auto *callee: Function * = call->getCalledFunction()) {
    if (callee->getName().starts_with(Prefix: lgcName::CooperativeMatrixTranspose)) {
      Value *src = call->getArgOperand(i: 0);
```

Problems

- New intrinsics are all CallInst
  - Common pattern: iterate over instructions, cast to CallInst, compare name string
  - Accessing arguments with an index

- Not developer-friendly

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def LgcCmDialect : Dialect {
  let name = "lgc.cooperative.matrix";
  let cppNamespace = "lgc::cooperativeMatrix";
}
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def LgcCmDialect : Dialect {
  let name = "lgc.cooperative.matrix";
  let cppNamespace = "lgc::cooperativeMatrix";
}
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def LgcCmDialect : Dialect {
  let name = "lgc.cooperative.matrix";
  let cppNamespace = "lgc::cooperativeMatrix";
}
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def LgcCmDialect : Dialect {
  let name = "lgc.cooperative.matrix";
  let cppNamespace = "lgc::cooperativeMatrix";
}
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);

  let summary = "transpose a matrix";
  let description = [{
    Given a cooperative matrix, the element type and the matrix layout, returns
    the transposed matrix.
  }];
}
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
```

```
CallInst *result = CreateNamedCall(funcName: callName, retTy: matrix->getType(),
                                   args, attribs: {[0]=Attribute::ReadOnly, [1]=Attribute::WillReturn});
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins FixedVectorType $element_type, $num_elements:$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);
```

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);
```

Many other constraints possible, see the Constraints.md in the llvm-dialects repo

Solution: llvm-dialects

```
4  define <8 x float> @test_transpose_simple(<8 x float> %a) {
5    %a.t = call <8 x float> @lgc.cooperative.matrix.transpose.v8f32(<8 x float> %a, i32 1, i32 0)
6    ret <8 x float> %a.t
7  }
```

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);

  let summary = "transpose a matrix";
  let description = [{
    Given a cooperative matrix, the element type and the matrix layout, returns
    the transposed matrix.
  }];
}
```

Solution: llvm-dialects

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);
```

Solution: llvm-dialects

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);
```

```
CoopMatrixTransposeOp *result = create<CoopMatrixTransposeOp>(
                                    matrix, elementType, layout);

std::string callName(s: lgcName::CooperativeMatrixTranspose);
Value *args[] = {[0]=matrix, [1]=elementType, [2]=layout};

CallInst *result = CreateNamedCall(funcName: callName, retTy: matrix->getType(),
                                   args, attribs: {[0]=Attribute::ReadOnly, [1]=Attribute::WillReturn});
```

Solution: llvm-dialects

```
def CoopMatrixTransposeOp : Op<LgcCmDialect, "transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);
```

```
if (isa<cooperativeMatrix::CoopMatrixTransposeOp>(Val: input)) {
```

```
if (auto *call: CallInst * = dyn_cast<CallInst>(Val: input)) {
  if (auto *callee: Function * = call->getCalledFunction()) {
    if (callee->getName().starts_with(Prefix: lgcName::CooperativeMatrixTranspose)) {
      Value *src = call->getArgOperand(i: 0);
```

Solution: llvm-dialects

```
CoopMatrixTransposeOp *result = create<CoopMatrixTransposeOp>(
                                    matrix, elementType, layout);
```

```
static const auto visitor =      You, 12 hours ago • Uncommitted changes
    llvm_dialects::VisitorBuilder<LowerCooperativeMatrix>()
        .add(fn: &LowerCooperativeMatrix::visitCooperativeMatrixTranspose)
        .build();

visitor.visit(&payload: *this, &: module);
```

```
void LowerCooperativeMatrix::visitCooperativeMatrixTranspose(
    cooperativeMatrix::CoopMatrixTransposeOp &coopTranspose) {
```

Solution: llvm-dialects

```
def CoopMatrixTransposeOp : LgcCmOp<"transpose", [Memory<[(read)]>, WillReturn]> {
  let arguments = (ins (FixedVectorType $element_type, $num_elements):$matrix, I32:$elem_type, I32:$layout);
  let results = (outs (eq $matrix):$result);
```

Solution: llvm-dialects

```
]> {
  I32:$elem_type, I32:$layout);
```

```
transpose.getElemType()
transpose.getLayout()
```

```
transpose->getArgOperand(i: 1)
transpose->getArgOperand(i: 2)
```

Additional features

- More complex constraints
- OpSet/OpMap working on classes of Operations
- ContextExtension

Future Work

- Custom Type Definitions
- Human readable structs in metadata

Thank you for listening!