第三部分：代码优化与生成

1. 内容简介

本次实验包含两部分，共6 分：

- 代码优化（2 分）： 消除Pony 程序中冗余的 transpose 函数
- 中间代码生成（4 分）： 将Pony 语言的**矩阵转置乘**（@，其语义为$A@B = AB^T$）操作转换到MLIR 的内置dialects，并最终生成LLVM 代码执行

开始本部分前请务必先用git pull 最新repo的lab3分支，同时注意保存之前部分的代码

2. 代码优化

2.1 功能实现

Pony 语言内置的transpose 函数会对矩阵进行转置操作。然而，对同一个矩阵进行两次转置运算会得到原本的矩阵，相当于没有转置。矩阵的转置运算是通过嵌套for 循环实现的，而嵌套循环是影响程序运行速度的重要因素。因此，侦测到这种冗余代码并进行消除是十分必要的。这里同学们需要在对应文件中根据提示补充优化pass 的关键代码，并测试是否真正实现了冗余代码消除。

文件地址：/pony_compiler/src/pony/mlir/PonyCombine.cpp

要求实现以下功能：

```
* 将pony dialect 的冗余转置代码优化pass 补充完整，最终实现冗余代码的消除。
```

注意事项：

- 在PonyCombine.cpp 搜索"TODO"，可以看到需要实现的相关函数以及具体要求

2.1 实验验证

在完成上述代码优化功能后，可以运行测试用例test_13。test_13 提供了一个冗余转置操作的实例，我们要求编译器能够通过代码优化去掉冗余的转置操作，输出优化后的结果，同学们可以对比优化前后的结果，从而更直观地理解冗余消除的效果。对于test_13 中的例子，分别执行以下命令，可以得到未优化和优化后的中间表示，结果如下图：

```
./bin/pony ../../test/test_13.pony -emit=mlir
```

```
root@e062ec9d43ba:/home/workspace/pony_compiler/src/build# ./bin/pony ../../test/test_13.pony -emit=mlir
module {
  pony.func private @transpose_transpose(%arg0: tensor<*xf64>) -> tensor<*xf64> {
    %0 = pony.transpose(%arg0 : tensor<*xf64>) to tensor<*xf64>
    %1 = pony.transpose(%0 : tensor<*xf64>) to tensor<*xf64>
    pony.return %1 : tensor<*xf64>
  }
  pony.func @main() {
    %0 = pony.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %1 = pony.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64>
    %2 = pony.generic_call @transpose_transpose(%1) : (tensor<2x3xf64>) -> tensor<*xf64>
    pony.print %2 : tensor<*xf64>
    pony.return
  }
}
```

```
./bin/pony ../../test/test_13.pony -emit=mlir -opt
```

```
root@e062ec9d43ba:/home/workspace/pony_compiler/src/build# ./bin/pony ../../test/test_13.pony -emit=mlir -opt
module {
  pony.func @main() {
    %0 = pony.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    pony.print %0 : tensor<2x3xf64>
    pony.return
  }
}
```

3. 代码优化

3.1 功能实现

在MLIR 中，高级语言会由高到低转换成不同抽象层级的中间表示（称为dialect），生成对应的中间代码，并最终生成最底层的可执行代码。为了执行一个Pony 语言的程序，我们需要以此：

1. 将Pony 程序（.pony）文件解析并生成对应的pony dialect 表示

2. 将pony dialect 转换成MLIR 内置的一些dialects （arith, memref 和affine）

3. 将affine dialect 转换成可被执行的llvm dialect

其中，第1 步我们已经支持基于前面两部分作业生成的AST，得到对应的pony dialect；第3 步从内置dialect 到llvm dialect 的转换也已由MLIR 本身支持。同学们只需关注第2 步，其中3 个内置的dialects 作用分别为：

- arith：负责代数运算操作，例如用 arith.constant 声明常数，用 arith.addf 和 arith.mulf完成浮点数的加和乘操作，详见该文档。

- memref：负责内存相关操作，例如用 memref.alloc 和 memref.dealloc 进行内存的分配和释放，详见该文档

- affine：负责循环相关操作，例如用 affine.for 进行循环遍历，用 affine.load 和 affine.store进行数据的读写，详见该文档

我们已经实现了pony dialect 中的大多数操作到内置dialects 的转换，以一个简单的pony 程序为例：

```
def main() {
  var a<2> = [1, 2];
  var b<2> = [3, 4];
  var c = a + b;
  print(c);
}
```

其对应的优化后的pony dialect 下图所示：

```
root@e062ec9d43ba:/home/workspace/pony_compiler/src/build# ./bin/pony ../../test/test.pony -emit=mlir
module {
  pony.func @main() {
    %0 = pony.constant dense<[1.000000e+00, 2.000000e+00]> : tensor<2xf64>
    %1 = pony.reshape(%0 : tensor<2xf64>) to tensor<2xf64>
    %2 = pony.constant dense<[3.000000e+00, 4.000000e+00]> : tensor<2xf64>
    %3 = pony.reshape(%2 : tensor<2xf64>) to tensor<2xf64>
    %4 = pony.add %1, %3 : (tensor<2xf64>, tensor<2xf64>) -> tensor<*xf64>
    pony.print %4 : tensor<*xf64>
    pony.return
  }
}
```

转换得到的内置dialects 表示如下图所示，该程序使用 arith.constant, memref.alloc 和 affine.store 初始化两个64 位浮点数组（%0 和%1，对应a 和b）并为结果数组（%3，对应c）分配空间，随后在 affine.for 内遍历两个数组，并使用 affine.load, arith.addf 和 affine.store 读取输入相加后存入结果数组。这里命令行中分别加入-emit=mlir, -emit=mlir-affine 和-emit=jit 即可执行相应级别的操作，方便大家进行实验调试。

```
root@e062ec9d43ba:/home/workspace/pony_compiler/src/build# ./bin/pony ../../test/test.pony -emit=mlir-affine
module {
  func @main() {
    %cst = arith.constant 4.000000e+00 : f64
    %cst_0 = arith.constant 3.000000e+00 : f64
    %cst_1 = arith.constant 2.000000e+00 : f64
    %cst_2 = arith.constant 1.000000e+00 : f64
    %0 = memref.alloc() : memref<2xf64>
    %1 = memref.alloc() : memref<2xf64>
    %2 = memref.alloc() : memref<2xf64>
    affine.store %cst_2, %2[0] : memref<2xf64>
    affine.store %cst_1, %2[1] : memref<2xf64>
    affine.store %cst_0, %1[0] : memref<2xf64>
    affine.store %cst, %1[1] : memref<2xf64>
    affine.for %arg0 = 0 to 2 {
      %3 = affine.load %2[%arg0] : memref<2xf64>
      %4 = affine.load %1[%arg0] : memref<2xf64>
      %5 = arith.addf %3, %4 : f64
      affine.store %5, %0[%arg0] : memref<2xf64>
    }
    pony.print %0 : memref<2xf64>
    memref.dealloc %2 : memref<2xf64>
    memref.dealloc %1 : memref<2xf64>
    memref.dealloc %0 : memref<2xf64>
    return
  }
}
```

在第二部分中，我们在Pony 语言中新增了二维**矩阵转置乘法**操作（@），其语义为$A@B = AB^T$，在pony dialect 中表示为pony.gemm，本次实验只需同学们实现 pony.gemm 到MLIR 内置dialects 的转换。

文件地址：/pony_compiler/src/pony/mlir/Dialect.cpp

/pony_compiler/src/pony/mlir/LowerToAffineLoops.cpp

要求实现以下功能：

实现Dialect.cpp 中的 GemmOp::inferShapes，推断矩阵乘操作结果的形状以进行内存分配；补全 LowerToAffineLoops.cpp 中的 GemmOpLowering，实现 pony.gemm 到MLIR 内置dialects 的转换。

注意事项：

- 在Dialect.cpp 和LowerToAffineLoops.cpp 搜索"TODO"，可以看到需要实现的相关函数以及具体要求
- 可以参考其他操作的转换完成本部分实验，例如Dialect.cpp 中的 MultOp::inferShapes 和 TransposeOp::inferShapes，和LowerToAffineLoops.cpp 中的 BinaryOpLowering

3.2 实验验证

我们以test_10 为例验证矩阵乘操作是否能被正确转换并执行：

```
$ cmake --build . --target pony
$ ./bin/pony ../../test/test_10.pony -emit=jit
```

如果执行结果如下图所示，表示转换并执行正确。

```
root@e062ec9d43ba:/home/workspace/pony_compiler/src/build# ./bin/pony ../../test/test_10.pony -emit=jit
14.000000 32.000000
32.000000 77.000000
```