MLIR 编译框架的使用与探索

第三部分：代码优化与生成

上海交通大学计算机系

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 语言的二维矩阵乘（`@`）操作转换到 MLIR 的内置 dialects，并最终生成 LLVM 代码执行

2　代码优化

在开始本次大作业前，请先在（docker 容器外的）`pony_compiler` 主目录下执行 `git pull` 拉取更新!!!

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/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

2.2　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，同学们可以对比下优化前后的结果，从而更直观地理解冗余消除的效果。详细步骤如下：对于 test_13 中的例子：

```
def transpose_transpose(x) {
  return transpose(transpose(x));
}
def main() {
  var a<2, 3> = [[1, 2, 3], [4, 5, 6]];
  var b = transpose_transpose(a);
  print(b);
}
```

执行以下指令，输出转换后的 pony dialect（即转换后的代码表示），查看输出结果可判断是否成功消除冗余转置

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

执行以下指令查看未优化的输出，对比优化前后输出的差异

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

3 中间代码生成

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 如图1所示；转换得到的内置 dialects 表示如图2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；最终的执行结果如图3所示。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@10438fcc71dd:/home/workspace/pony_compiler/build# ./bin/pony ../test/test.pony -emit=mlir -opt
module {
  pony.func @main() {
    %0 = pony.constant dense<[1.000000e+00, 2.000000e+00]> : tensor<2xf64>
    %1 = pony.constant dense<[3.000000e+00, 4.000000e+00]> : tensor<2xf64>
    %2 = pony.add %0, %1 : tensor<2xf64>
    pony.print %2 : tensor<2xf64>
    pony.return
  }
}
```

图 1. 示例程序的 pony dialect 表示

```
root@10438fcc71dd:/home/workspace/pony_compiler/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
  }
}
```

图 2. 示例程序的内置 dialects 表示

在第二部分中，我们在 Pony 语言中新增了二维矩阵乘法操作（@），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/pony/mlir/Dialect.cpp 和
/pony_compiler/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`

```
root@10438fcc71dd:/home/workspace/pony_compiler/build# ./bin/pony ../test/test.pony -emit=jit
4.000000 6.000000 root@10438fcc71dd:/home/workspace/pony_compiler/build#
```

图 3. 示例程序的运行结果

3.2 实验验证

```
root@10438fcc71dd:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_10.pony -emit=jit
22.000000 28.000000
49.000000 64.000000
```

图 4. test_10 的运行结果

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

如果执行结果如图4所示，表示转换并执行正确。

4 提交内容

请在截止日期前在 Canvas 平台指定位置提交大作业的压缩文件，应包含一份实验报告和代码。
1）实验报告：
注意事项：

- 此次实验报告中需要附带**实验一、二、三所有测试用例**（test_1 到 test_13）的测试结果，且必须以**“指令 + 输出结果”完整截图**的形式展示；
- 提交截止日期为 **6 月 23 日**。

除此之外，本次实验报告应指出三次实验的实验过程、实验结果，并能完整、准确地说出自己的设计思路，描述清楚各个函数的作用、具体实现方法等。总结部分可以分享自己在实验过程中遇到的问题和解决方法，对 MLIR 框架的认识。也可以对大作业提出自己的意见，我们将在后面的学期不断进行完善。**实验报告中应写明同学的姓名、学号、邮箱等联系方式**。
2）代码：

请将实验一、二、三全部修改代码对应的文件、实验结果的截图（按照对应测试用例 test_n 命名）一起打包提交。如果你在 TODO 以外的地方修改代码，请在你修改/添加代码的位置加上注释（例如第一部分：// the first part），方便我们后续审核。