基于 MLIR 的 SysY 2022 语言扩展实现

一、 MLIR 介绍

MLIR 是 Google 在 2019 年推出的编译器设施。传统的编译器会提供一系列预先定义好的指令和数据类型；而 MLIR 提供了方言 (dialect) 使得用户能够自定义各式各样的操作 (operation)··、属性 (attribute) 以及数据类型 (type)，表现出了更强的可用性和阔扩展性。在 MLIR 的语境下，操作是广义的 IR 指令，它的操作数满足 SSA 性质。

除了用户自定义的方言之外，MLIR 还自定义了一系列通用的方言，并实现了相关的数据流分析和优化框架，如 affine、scf、memref 等。MLIR 框架鼓励用户以逐层降级的方式将自定义的方言降级到 MLIR 预先定义的方言中并使用相关的优化设施来完成优化。

1.1 Affine 方言

MLIR affine 方言提供了表示具有静态仿射约束 (SCoP, Static Control Parts) 性质的嵌套循环的操作来进行多面体模型建模、分析和优化。一个 affine.for 表示一个以上下界都是循环不变量的循环，而 affine.parallel 则表示并行循环。进行数据的读取和写回操作时需要使用 affine.load 和 affine.store 操作。

1.2 MemRef 方言

MLIR 提供了内建的数据结构 memref 用于表示类似于 C 语言中裸指针的概念，并在 memref 方言中提供了作用于该数据结构的操作。图 1 所示的代码片段展示了一个由 affine 方言和 memref 方言实现的用于计算 C[i + j] += $A[i] \cdot B[j]$的 MLIR 代码片段：

```
%c0 = arith.constant 0 : index
%0 = memref.dim %A, %c0 : memref<?xf32>
%1 = memref.dim %B, %c0 : memref<?xf32>
affine.for %i = 0 to affine_map<()[s0] -> (s0)>()[%0] {
  affine.for %j = 0 to affine_map<()[s0] -> (s0)>()[%1] {
    %2 = affine.load %A[%i] : memref<?xf32>
    %3 = affine.load %B[%j] : memref<?xf32>
    %4 = arith.mulf %2, %3 : f32
    %5 = affine.load %C[%i + %j] : memref<?xf32>
    %6 = arith.addf %4, %5 : f32
    affine.store %6, %C[%i + %j] : memref<?xf32>
  }
}
```

图 1 affine 方言和 memref 方言的代码示例

1.3 其它相关方言

- SCF (Structured Control Flow) 方言：提供了结构化地表示循环和条件语句的操作，但是对于边界条件没有 SCoP 性质的约束
- Arith 方言：提供了各种数学计算相关的操作
- OpenMP 方言：提供了平台无关的 OpenMP 描述，提供了将并行循环表示到 OpenMP 运行时的中间层
- Vector 方言：提供了平台无关的向量操作

- LLVM 方言：MLIR 中所有方言的最终降级目标，用于生成 LLVM IR

二、 SysY 2022 语言的扩展

2.1 Sysy MLIR 的编译流程

本项目的 SysY 2022 语言编译器（**Sysy P**arallel **C**ompiler，此后简记为 sysypc）使用 ANTLR 4 实现了前端，完成源代码到 AST 的转换。之后 sysypc 会利用 MLIR 逐层降级的能力对 SysY 2022 的扩展语法和非扩展语法部分使用不同的降级路线对代码进行优化并生成 LLVM IR 以及可重定位目标文件。最后使用普通的链接器完成 SysY 和 OpenMP 运行时库链接生成可执行目标程序。图 2 展示 sysypc 的编译流程。

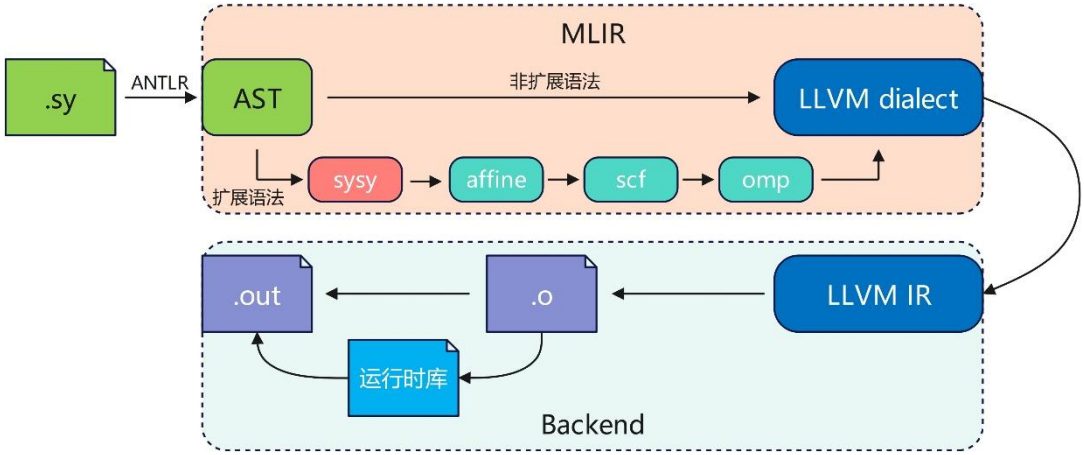

图 2 Sysypc 的编译流程

2.2 使用 ANTLR 实现前端

略

2.3 从 AST 降级到 MLIR

2.3.1 Sysy 方言中的操作

Sysy 2022 语言扩展提供了基于张量的并行计算能力。因此第一类操作是用于声明和初始化张量数据的 InitTensorOp 和 DeclTensorOp，以及直接创建张量对象的 ConstantOp。

第二类操作定义了张量的逐元素计算，也可以认为是在编译器层面进行了针对张量的运算符重载。这类操作包括：AddOp、SubOp、MulOp 和 DivOp。

除了基础的逐元素计算操作之外，为了提高我们扩展的张量数据结构的可用性，我们还提供了一系列基于张量类型的内建函数。这些内建函数并不需要在链接时和特定的库链接，它们会在降级过程中完成代码生成。这些内建函数有表示矩阵乘法 MatMulOp、表示卷积的 ValidConvOp 和 SameConvOp、对张量进行转置的 TransposeOp、进行张量间复制的 CopyOp 以及几个规约操作 MaxOp、MinOp 和 SumOp。

由于在扩展语法中张量类型被视为一等对象，因此它应该能够作为函数的参数类型、进行数据类型的转换。所以我们在 sysy 方言中定义了 FunctionOp 处理函数声明、RetOp 处理返回语句、GenericCallOp 处理函数调用以及 CastOp 处理类型转换。

最后还有一些辅助函数，如打印张量的 PrintOp。这些辅助函数并没有扩展 Sysy 2022 的运行时库，仍旧是在降级时转换为普通的运行时库调用

（PrintOp 和 ClearOp 的功能是什么？不太清楚，记得补上）

完整清单如下：

- 创建张量相关：
  - InitTensorOp
  - DeclTensorOp
  - ConstantOp
- 张量的逐元素计算：
  - AddOp
  - SubOp
  - MulOp
  - DivOp
- 张量的内建函数：
  - MatmulOp
  - ValidConvOp
  - SameConvOp
  - TransposeOp
  - CopyOp
  - MaxOp
  - MinOp
  - SumOp
- 处理被视为一等对象的张量对象：
  - FunctionOp
  - RetOp
  - GenericCallOp
  - CastOp
- 辅助函数：
  - PrintOp
  - ScanOp
  - ClearOp

2.3.2 Sysy 方言中操作的降级和优化

为了从我们定义的 sysy 方言中生成 LLVM IR，我们需要利用 MLIR 中提供的基础设施进行降级。在降级之外，我们还能够利用 MLIR 中提供的组件进行代码优化。本小节将以 MatMulOp 为例介绍从 sysy 方言到 LLVM 方言的降级和优化过程。

在没有降级前的 MatMulOp 如图 4 所示：

```
sysy.matmul %C, %A, %B : tensor<16x16xf32>, tensor<16x16xf32>, tensor<16x16xf32>
```

图 4 未降级的 MatMulOp

由于 tensor 只是对数据结构的表示而没有指向资源的句柄，所以需要将 tensor 类型降级为 memref 类型，如图 5 所示：

```
sysy.matmul %C, %A, %B : memref<16x16xf32>, memref<16x16xf32>, memref<16x16xf32>
```

图 5 降级操作数的 MatMulOp

在之后的降级过程中，利用 MLIR 中的 affine 方言的 vector 方言描述循环和向量操作，进行了初步的优化：

```
affine.for %arg0 = 0 to 16 {
  affine.for %arg1 = 0 to 16 step 4 {
    affine.for %arg2 = 0 to 16 {
      %11 = affine.load %A[%arg0, %arg2] : memref<16x16xf32>
      %12 = vector.splat %11 : vector<4xf32>
      %13 = vector.load %B[%arg2, %arg1] : memref<16x16xf32>, vector<4xf32>
      %14 = vector.load %C[%arg0, %arg1] : memref<16x16xf32>, vector<4xf32>
      %15 = vector.fma %12, %13, %14 : vector<4xf32>
      vector.store %15, %C[%arg0, %arg1] : memref<16x16xf32>, vector<4xf32>
    }
  }
}
```

图 5 降级到 affine 方言和 vector 方言的 MatMulOp

通过利用 affine 方言中的基础设施，我们得以对 affine.for 表示循环进行分块和并行化来进一步提升性能：

```
affine.parallel (%arg0) = (0) to (16) step (4) {
  affine.for %arg1 = 0 to 16 step 16 {
    affine.for %arg2 = 0 to 16 step 16 {
      affine.for %arg3 = affine_map<(d0) -> (d0)>(%arg0) to affine_map<(d0) -> (d0 + 8)>(%arg0) {
        affine.for %arg4 = affine_map<(d0) -> (d0)>(%arg1) to affine_map<(d0) -> (d0 + 16)>(%arg1) step 4 {
          affine.for %arg5 = affine_map<(d0) -> (d0)>(%arg2) to affine_map<(d0) -> (d0 + 16)>(%arg2) {
            %11 = affine.load %A[%arg3, %arg5] : memref<16x16xf32>
            %12 = vector.splat %11 : vector<4xf32>
            %13 = vector.load %B[%arg5, %arg4] : memref<16x16xf32>, vector<4xf32>
            %14 = vector.load %C[%arg3, %arg4] : memref<16x16xf32>, vector<4xf32>
            %15 = vector.fma %12, %13, %14 : vector<4xf32>
            vector.store %15, %C[%arg3, %arg4] : memref<16x16xf32>, vector<4xf32>
          }
        }
      }
    }
  }
}
```

图 6 优化后的 affine 循环

之后可以利用 MLIR 中已经封装好的 pass 将这些 MLIR 已有的方言表示并行结构和控制流等，然后降级到 LLVM 方言上，最终生成可重定位目标文件和运行时库链接。