Lecture one

Introduction to MLIR and LLVM

HPC motivation

- In HPC we rely heavily on libraries and compilers, and a number of these have already been ported on-top of MLIR, with more to follow
- But we need to be realistic here, MLIR was only created in 2022, so whilst it is growing very rapidly, it is still developing and being enhanced
  - This is one of the exciting aspects, as we in the HPC community have the opportunity to engage and influence its development
  - The ecosystem has matured significantly in the last 18 months, so it is now a realistic opportunity to build on-top of this and to expect benefits

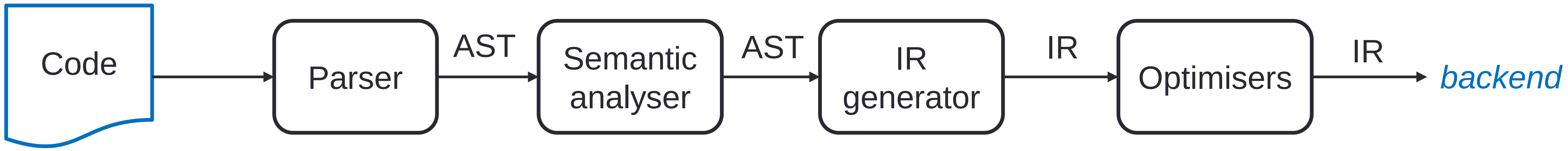
Technology overview

- MLIR is a framework for developing your own compiler Intermediate Representation (IR)
- Allowing different levels of abstraction to co-exist
  - Progressive lowering through these levels during compilation from one dialect to another
  - You the compiler/tool developer choose where these abstraction layers start & stop

- LLVM is a collection of modular compiler technologies that aids reuse
  - Contains the clang, flang etc compilers
- For our uses, the most interesting part are the backends which target different architectures
- We can interact with these by generating LLVM-IR

The challenge MLIR is looking to solve

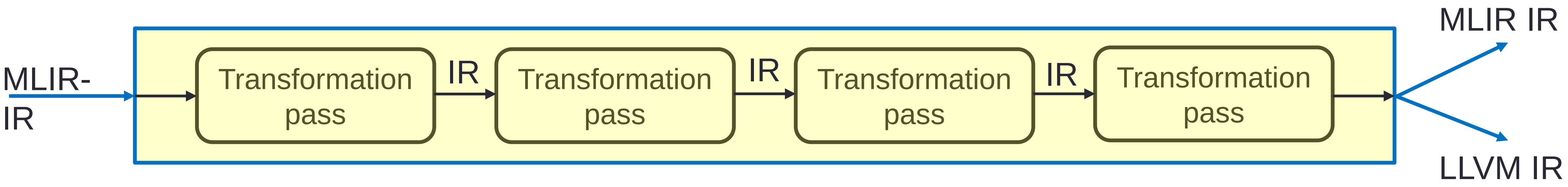
- Whilst LLVM has become very popular, much of the reuse is in the backend where LLVM-IR can target different architectures



- However, activities in the frontend can be extremely time consuming
  - Especially designing the IR, generating it and then and the optimisers
- This is where MLIR fits, providing a framework for defining and manipulating IRs
  - As well as many existing IRs and transformations to promote reuse

Executing MLIR

- LLVM provides the *mlir-opt* tool which accepts inputs and generates outputs
  - Can input MLIR format IR, output can be MLIR IR, LLVM IR etc
  - Can instruct the tool to undertake passes on the IR



- Other tools wrap this, such as Flang which is LLVM's Fortran compiler
  - Parses the Fortran code into it's own MLIR dialect (FIR) with other optional dialects, then undertakes transformation on the code before generating the object files

MLIR-IR is human readable

- MLIR-IR can be viewed at each stage and is (fairly!) easy to read
  - Can be useful for debugging

```
func.func(%arg0) ({
 %x = arith.constant() {"value" = 10 : i32} : () -> i32
 %y = arith.addi(%x, %arg0) : (i32, i32) -> i32
 func.return(%y) : (i32) -> i32
}) {"sym_name" = “example“} (i32) -> (i32)
```

- This is Single Static Assignment (SSA) form, we will see this lots in the tutorial
- SSA comprises operations which have any number of operands (arguments), return zero, one or more results, and can contain static attributes
  - You can think of these a bit like function calls but translated by the compiler rather than each calling a function.

MLIR-IR is the ecosystem

- MLIR provides the overarching framework here but known nothing about specific instructions

```
%x = "any_unknown_operation_here"(%arg0, %arg0) : (i32, i32) -> i32
%y = "my_increment"(%x) : (i32) -> i32
```

- These will be caried through the transformation passes and it is expected that some of the transformations know how to handle these
- Therefore MLIR is an open, highly pluggable ecosystem where new dialects and new transformations can be created without having to modify others

- But there are standard dialects provided which include standard types

Mixing dialects

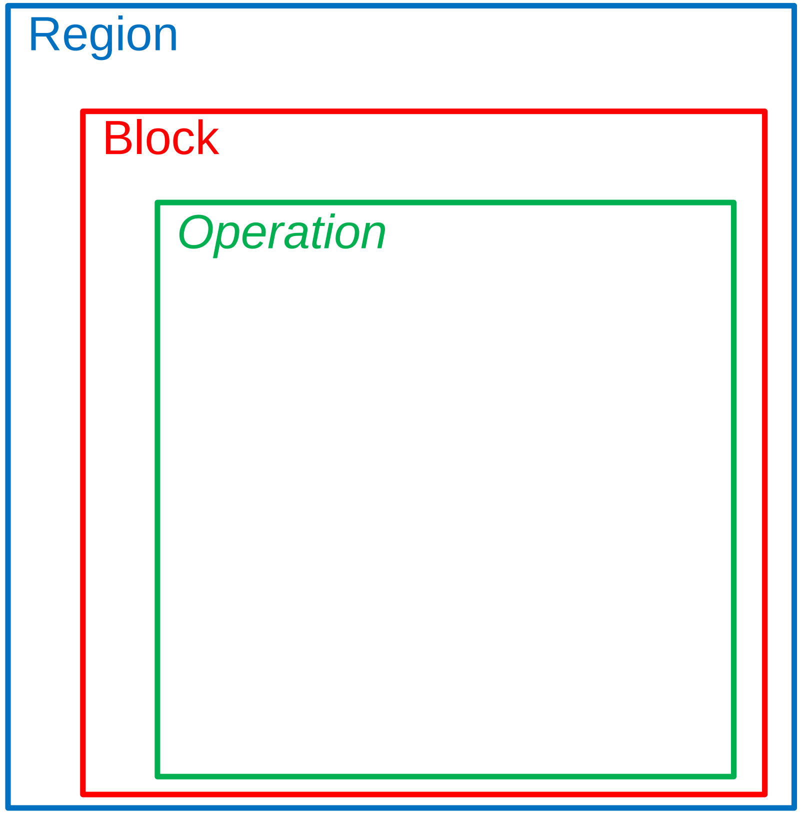
```
func.func(%arg0) ({
 %x = arith.constant() {"value" = 10 : i32} : () -> i32
 %y = arith.addi(%x, %arg0) : (i32, i32) -> i32
 func.return(%y) : (i32) -> i32
}) {"sym_name" = "example"} (i32) -> (i32)
```

- In this example we are already seeing some of MLIR's standard dialects being mixed and matched
  - The *func* dialect which marks this as a function and the return
  - The *arith* dialect that undertakes the integer addition (*addi)* and defines the constant (*constant)*

- Could add our own dialects and include them here too

MLIR key concepts…

- MLIR is based on a graph style data structure of nodes which are called *Operations* and edges called *values*.
- Regions contain an ordered set of blocks, and blocks contain an ordered set of operations
  - Operations can also contain regions, and hence hierarchical structures can be expressed
- Operations are where most of the definition occurs
  - Can represent pretty much anything; functions definitions, function calls, variable allocation, conditionals, assignment etc
  - Operations can be arbitrarily extended



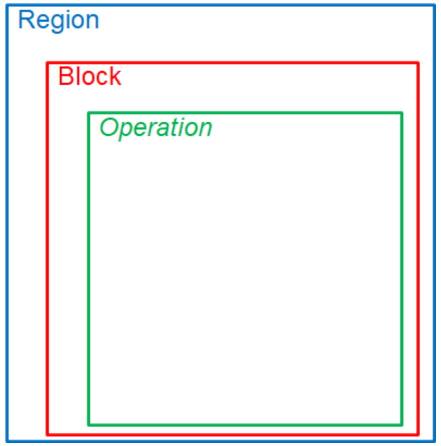
ExCALIBUR
xDSL

Operations and their children

```
func.func(%arg0) ({
 %x = arith.constant() {"value" = 10 : i32} : () -> i32
 %y = arith.addi(%x, %arg0) : (i32, i32) -> i32
 func.return(%y) : (i32) -> i32
}) {"sym_name" = "example"} (i32) -> (i32)
```

- The *func* operation (from the *func)* dialect contains a region with a single block that itself contains a series of operations (the *constant, addi*, and *return* operations)
- The *func* operation also has a string attribute (*sym_name*) which is set during compilation and an operand argument (*arg0*).

Blocks



- Blocks contain a list of operations and can be thought of as executing these in order
- Blocks accept an optional list of arguments similar to a function

```
^bb0(%0: i32):
  %1 = arith.constant() {"value" = 10 : i32} : () -> i32
  %2 = arith.cmpi(%0, %1) {“predicate” = 2 : i32} : (i32, i32) -> i1
  cf.cond_br(%2) [^bb1, ^bb2]  { "operand_segment_sizes" = array<i32: 1, 0, 0> }: (i1)

^bb1():
  cf.br() [^bb2] : () -> ()

^bb2():
  ……
```

Blocks

A block has zero, one or more arguments

Compare integers %0 and %1 for less than (2 passed to predicate) providing a 1 bit integer value (Boolean)

```
^bb0(%0: i32):
  %1 = arith.constant() {"value" = 10 : i32} : () -> i32
  %2 = arith.cmpi(%0, %1) {"predicate" = 2 : i32} : (i32, i32) -> i1
  cf.cond_br(%2) [^bb1, ^bb2]  { "operand_segment_sizes" = array<i32: 1, 0, 0> }: (i1)

^bb1():
  cf.br() [^bb2] : () -> ()

^bb2():
  ……
```

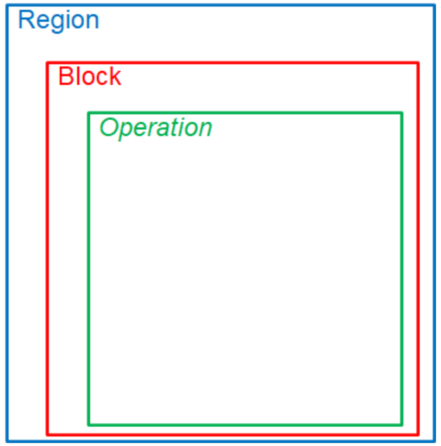
If true then jump to the ^*bb1* block, otherwise jump to the ^*bb2* block

Unconditional branch to block ^*bb2*

ExCALIBUR

xDSL

Regions



- A region contains an ordered list of blocks

- Regions must be contained within operations, therefore they are hierarchical
  - The body of a function is an example of a region, contained within the function definition operation
- Regions also provide encapsulation, where one can only branch to blocks within the same region

Types

- MLIR provides its own type system and is strongly typed
  - MLIR provides the framework and the dialects provide the actual types themselves
    - Any number of types can be defined by dialects

```
func.func(%arg0) ({
 %x = arith.constant() {"value" = 10 : i32} : () -> i32
 %y = arith.addi(%x, %arg0) : (i32, i32) -> i32
 func.return(%y) : (i32) -> i32
}) {"sym_name" = "example"} (i32) -> (i32)
```

- Can also provide aliases
  - i.e. i32 is an alias for the IntegerType of width 32
  - f64 is an alias for the FloatType of width 64
  - Some types accept parameters – such as width and kind (useful for Fortran)

Defining dialects

- Dialects are the mechanism by which the MLIR ecosystem is extended
- Can define new operations, new attributes, and new types
  - Each attribute has a unique name and this is prefixed to each of these
- Dialects are consumed by passes, with conversions between dialects

- Standard dialects provided as part of MLIR are:

  - acc
  - affine
  - amdgpu
  - amx
  - arith
  - arm_neon
  - arm_sve
  - async
  - bufferization
  - cf
  - complex
  - dlti
  - emitc
  - func
  - gpu
  - index
  - linalg
  - LLVM
  - math
  - memref
  - ml_program
  - nvgpu
  - nvvm
  - omp
  - pdl
  - pdl_interp
  - quant
  - rocdl
  - scf
  - shape
  - sparse_tensor
  - tensor
  - vector
  - x86vector
  - Builtin
  - SPIR-V
  - TOSA
  - Transform

Full details on each of these is at https://mlir.llvm.org/docs/Dialects

Example flow……

- We will see this in more detail in the hands-on activities, but to give you a feeling for it now

```
user@login01:~$ mlir-opt --convert-func-to-llvm ir.mlir | mlir-translate -mlir-to-llvmir | clang -x ir -o test -
```

Provide an MLIR file as input to the *mlir-opt* call which provides numerous transformations

Once in its final form the *mlir-translate* tool will convert from MLIR to a number of output formats, here we are generating LLVM IR

The standard LLVM IR is then provided to Clang which will compile it to an object file or executable

- But where do we get the MLIR input file (here *ir.mlir*) in the first place?
  - That's the job of our language specific frontend and we will see this in more detail in the hands on activities

Conclusions

- MLIR is a very powerful framework, promoting reuse of infrastructure especially because it is backed by LLVM
  - You often see it presented as *doing properly* what from experience with LLVM they wish they had changed with that framework!
  - This is probably a little unfair to LLVM, but certainly it is based on all the lessons learnt
- We have focussed on the overarching concepts in this lecture
  - Regions, which contain blocks, which contain operations – which themselves can contain regions.
  - The ability to define dialects and transformations, and for these to coexist together
  - The fact that numerous standard dialects and transformations are provided *out-of-the-box*