Unraveling MLIR: From High Level to LLVM with Neural Networks and Optimizations

Rafael Sumitani[1], **Guilherme Silva**[1], **and Michael Canesche**[1]

[1] Cadence Design Systems – Brazil

Abstract. *MLIR (Multi-Level Intermediate Representation) is an infrastructure for representing and transforming programs at multiple levels of abstraction. In this tutorial, we will explore how MLIR can be used to compile and optimize representations of computational graphs, such as those found in neural networks. The contribution of this work includes presenting the main concepts about MLIR, how to build and run it. Also, we teach you how to create MLIR passes.*

1. Introduction

MLIR (Multi-Level Intermediate Representation) [3] is a compiler infrastructure designed to support multiple levels of abstraction within a single framework. Its primary purpose is to enable the development of modular and reusable compiler components that can handle the various needs of modern computing, including machine learning [4, 5], high-performance computing [1], and domain-specific languages [2]. By allowing developers to define custom dialects and transformations, MLIR provides a flexible way to represent and optimize code across different stages of compilation, from high-level operations down to low-level machine instructions.

Modern compiler design faces challenges due to the increasing heterogeneity of hardware and the complexity of software stacks. Traditional compilers typically rely on a single intermediate representation (IR), which can limit their ability to perform optimizations across different abstraction levels or adapt to new hardware architectures. MLIR addresses this problem by introducing a multi-level IR system that supports a wide range of transformations and analyses at various levels of abstraction. This approach improves the ability to target specialized hardware, facilitates better integration of domain-specific optimizations, and promotes code reuse across different compiler projects.

MLIR was initially developed by the TensorFlow team at Google to address the limitations of the existing compiler infrastructure to handle machine learning workloads [1]. Recognizing the broader applicability of its design, MLIR was later integrated into the LLVM ecosystem, where it now serves as a foundational component for building modern, scalable, and extensible compilers. Its integration into LLVM has helped unify efforts across different compiler communities, fostering innovation and collaboration in the development of next-generation compiler technologies.

The Complexity of Modern Hardware and Software Stacks. In recent years, the landscape of computing has undergone a dramatic transformation. The rise of heterogeneous hardware—such as GPUs, TPUs, FPGAs, and custom accelerators—has introduced a level of complexity that traditional compiler infrastructures struggle to manage.

[1] https://www.modular.com/blog/democratizing-ai-compute-part-8-what-about-the-mlir-compiler-infrastructure

At the same time, software stacks have become increasingly layered and domain-specific, particularly in areas like machine learning, scientific computing, and real-time systems. These domains often require specialized optimizations and tight integration with hardware capabilities, which are difficult to express and manage using conventional compiler architectures.

This complexity is compounded by the need to support multiple programming languages, frameworks, and execution models. For example, a modern machine learning pipeline might involve Python for orchestration, TensorFlow or PyTorch for model definition, and highly optimized C++ or CUDA kernels for execution. Bridging these layers efficiently requires a compiler infrastructure that can operate across multiple levels of abstraction while maintaining performance and correctness. Traditional intermediate representations (IRs), such as LLVM IR or Java bytecode, are typically designed for a single level of abstraction, making them ill-suited for this kind of cross-domain, multi-level optimization.

MLIR: Enabling Modularity, Reuse, and Extensibility. MLIR was created to address these challenges by introducing a new approach to compiler design. Rather than relying on a single, rigid IR, MLIR provides a framework for defining multiple IR dialects, each tailored to a specific level of abstraction or domain. These dialects can interoperate within a unified infrastructure, allowing developers to build modular and reusable compiler components. This modularity enables more effective optimization pipelines, where high-level transformations can be applied before lowering to more hardware-specific representations.

One of MLIR's key strengths is its extensibility. Developers can define custom operations, types, and transformation passes without having to modify the core infrastructure. This makes it easier to experiment with new programming models, target emerging hardware, or integrate domain-specific optimizations. MLIR also supports advanced features such as affine loop analysis, pattern-based rewriting, and structured control flow, which are essential to optimize modern workloads. By providing a common infrastructure for these capabilities, MLIR reduces duplication of effort and fosters collaboration across compiler projects.

Comparison with Traditional Intermediate Representations. LLVM IR was designed for specific purposes, but has limitations when it comes to representing and optimizing across different levels of abstraction. LLVM IR is too low-level for capturing high-level semantics, such as those in machine learning frameworks. MLIR addresses these issues by supporting a hierarchy of intermediate representations (IRs) tailored to various stages of the compilation process. This enables more expressive and efficient transformations and allows different teams to work independently on different parts of the compiler. As a result, MLIR marks a major advancement in compiler infrastructure, offering a more scalable and adaptable foundation for modern software development. Examples 1 and 2 show in detail the difference between these two IRs.

Example 1 *Simple LLVM IR program which either returns a constant value or the result of a function call.*

```
define i32 @foo(i32 %0, i32 %1) {
  %3 = icmp slt i32 %0, %1
```

```
  br i1 %3, label %4, label %6
4:
  %5 = call i32 @bar(i32 %0, i32 %1)
  br label %7
6:
  br label %7
7:
  %8 = phi i32 [ 42, %6 ], [ %5, %4 ]
  br label %9
9:
  ret i32 %8
}
```

Example 2 *MLIR version of the LLVM IR program in Example 1. This representation uses high-level dialects such as* `scf`*. This allows to reason about concepts such as scopes, which are not available in LLVM IR.*

```
func.func @foo(%arg0: i32, %arg1: i32) -> i32 {
  %lt = arith.cmpi slt, %arg0, %arg1 : i32
  %res = scf.if %lt -> i32 {
    %0 = func.call @bar(%arg0, %arg1) : (i32, i32) -> i32
    scf.yield %0 : i32
  } else {
    %1 = func.call @bar(%arg1, %arg0) : (i32, i32) -> i32
    scf.yield %1 : i32
  }
  return %res : i32
}
```

In the sections that follow, we discuss MLIR's architecture and main concepts (Section 2); how to build it locally (Section 3); and a hands-on guide on how to build a custom pass (Section 4).

2. MLIR Architecture and Concepts

MLIR is a compiler infrastructure designed to support the development of reusable and extensible compiler components. It introduces a flexible IR system that allows multiple abstraction levels to co-exist, enabling better optimization and code transformation. MLIR is modular, supporting custom dialects and operations tailored to specific domains. It plays a key role in modern compiler stacks like LLVM and TensorFlow. Its architecture promotes scalability, reusability, and interoperability across diverse hardware and software targets. In this section, we will present the MLIR architecture and its main concepts. Example 3 illustrates some of these concepts with an abstract representation.

Example 3 *This MLIR example defines a module containing a single function named* `@name_func`*, which takes two input arguments of type* `tensor<2xi32>`*—representing 1D tensors with two 32-bit integers—and returns a tensor of the same type. Inside the function body, the operation* `my_dialect.my_operation` *is invoked, which belongs to a custom dialect*

named `my_dialect`. This operation takes the two input tensors and produces a result stored in `%0`. The function then returns this result using the `return` operation. The function body itself is a region, and the sequence of operations within it forms a block. This example showcases key MLIR concepts: dialects, operations, types, attributes, and the structural hierarchy of regions and blocks used to organize control flow and computation.

```
// Abstract example
module {
  func.func @name_func(%arg0: tensor<2xi32>, %arg1: tensor<2xi32
     >) -> tensor<2xi32> {
    %0 = my_dialect.my_operation %arg0, %arg1 {my_attr = "attr"}
        : tensor<2xi32>
    return %0 : tensor<2xi32>
  }
}
```

2.1. Core Concepts

MLIR is built on a few fundamental concepts that form its core structure. At its heart, MLIR organizes the code into dialects that define custom operations, types, and attributes. These elements are structured within regions and blocks, providing a flexible and hierarchical way to represent computations. By understanding these key components, including dialects, operations, types, attributes, regions, and blocks, we can unlock MLIR's power in optimizing and transforming code across different levels of abstraction.

2.1.1. Dialects

Dialects are modular and extensible namespaces that define a set of operations, types, and attributes tailored to a specific domain or abstraction level. Dialects can be created and reused, and multiple dialects from different levels of abstractions can be used together. This synergy between different levels of abstraction is essential to the MLIR ecosystem and it's part of what makes it so powerful. These are some of the dialects built-in into MLIR:

- **func**: This dialect encompasses operations related to function abstractions, such as definitions and calls. It can be used to greatly organize and modularize the IR.
- **tensor**: This dialect includes operations for creating and manipulating tensors. Tensors are a representation of multi-dimensional arrays that is commonly used in the context of machine learning. In the context of MLIR, tensors are high-level abstractions and they are not concerned with details such as data layout or memory addresses.
- **linalg**: This dialect provides a framework for representing high-level linear algebra computations, while providing useful transformations and optimizations. It also has many built-in commonly used operations, such as matrix multiplications and dot products.
- **arith**: This dialect contains basic integer and floating-point mathematical operations which can be applied to numeric scalars, tensors and vectors.

- **scf**: This dialect contains operations to represent structured control flow constructs, such as `for` loops and `if` statements.
- **affine**: This dialect offers a framework for using polyhedral compilation for different analyses and loop transformations.
- **memref**: This dialect provides operations for dealing with concrete memory buffers, which are represented by the `memref` type. Unlike tensors, this type will contain relevant information regarding the data's representation in memory, such as its data layout and its memory location.
- **gpu**: This dialect provides level device-agnostic abstractions for launching GPU kernels. It can be used as a middle stage before being lowered to device-specific GPU dialects, such as `nvvm`.

2.1.2. Operations

Operations are the fundamental units of computation in MLIR, analogous to LLVM instructions but with added flexibility. Unlike LLVM instructions, MLIR operations belong to a dialect namespace and can include type annotations, enabling richer semantic expression. Operations in MLIR can take any number of operands, and they can also have any number of results. A concrete example is shown in Example 4 which we use the `arith.muli` operation, which multiplies two floating point numbers.

Example 4 *This code shows an operation, `arith.muli`, which represents a multiplication between two 32-bit floating-point values. The operands `%lhs` and `%rhs` will be multiplied and the result is stored in `%res`.*

```
// A multiply operation in the 'arith' dialect
%res = arith.muli %lhs, %rhs : f32
```

2.1.3. Types

A **type** is a classification that specifies the kind of value that a variable can hold, and the operations that can be performed on it. Types help enforce data constraints, ensuring that operations are performed on compatible values.

There are many built-in types in MLIR. Some of them are scalars (Example 5), but there are also more complex types to represent structured data such as tensors (Example 6).

Example 5 *Primitive Types: Basic scalar types (defined in the builtin dialect).*

```
// Integer types (signed, unsigned, signless)
i32          // 32-bit signless integer
si8          // 8-bit signed integer
ui64         // 64-bit unsigned integer

// Floating-point types
f32          // 32-bit float
f64          // 64-bit double
```

```

// Index type (for loop counters/strides)
index       // Platform-dependent integer (e.g., size_t)
```

Example 6 *Tensor types: Used for multi-dimensional arrays (from the tensor dialect).*

```
// Ranked tensor
tensor<4x3x2xf32>          // 4x3x2 tensor of f32 elements
tensor<?x8xi64>            // [dynamic]x8 tensor of i64

// Unranked tensor
tensor<*xf32>             // Tensor of any rank with f32 elements

// Tensor with affine layout (e.g., for striding)
tensor<4x3xf32, affine_map<(d0, d1) -> (d0, d1)>>
```

2.2. Attributes

Attributes are used to represent data about an operation that are known at compile time, such as constant values (Example 7), string metadata (Example 8), or properties that do not change during execution. Each operation has an attribute dictionary, which associates a set of attribute names to attribute values. T

Example 7 *The dense attribute is a built-in MLIR attribute used to represent a collection of values.*

```
 %0 = "emitc.constant"(){value = dense<[[0.0, 1.0], [2.0, 3.0]]>
     : tensor<2x2xf32>} : () -> tensor<2x2xf32>
```

Example 8 *The string attribute does not affect computation, but can be used by passes or tools that interpret these metadata.*

```
%0 = "emitc.call_opaque"() {callee = "foo"} : () -> i32
```

2.2.1. Regions and Blocks

In MLIR, **regions** and **blocks** are core structural elements that define the control flow and nesting of operations. A **region** is a container that holds one or more **blocks** and is used to represent constructs such as function bodies, loop bodies, or conditional branches. Each block within a region is a linear sequence of operations with a single entry point and no internal control flow. Blocks can have arguments, which are values passed into them from the enclosing operation or from other blocks. Operations may also contain nested regions, thus enabling hierarchical structures that can represent structured control flow and scopes.

Example 9 *In the `scf.for` loop from the `scf` dialect, the loop body is a region containing a block that executes on each iteration. This design allows MLIR to model the structured control flow in a flexible and extensible way, supporting transformations and analyses across dialects.*

```
func.func @loop_example(%arg0: index) -> i32 {
  %init = arith.constant 0 : index
  %step = arith.constant 1 : index
  %sum_init = arith.constant 0 : i32
  // scf.for loop: iterates from 0 to %arg0 with step 1
  %result = scf.for %i = %init to %arg0 step %step
                iter_args(%sum = %sum_init) -> (i32) {
    // This is the body region of the loop
    // It contains a single block with three operations
    %val = arith.constant 1 : i32
    %sum_next = arith.addi %sum, %val : i32
    scf.yield %sum_next : i32
  }
  return %result : i32
}
```

Figure 1 illustrates the hierarchical structure of a function in MLIR, emphasizing how regions and blocks are organized in Example 9. At the top level, a `func.func` operation contains a region that includes block 1. This block holds three `arith.constant` operations and an `scf.for` loop, which itself introduces a nested region. Inside this nested region is block 2, comprising another `arith.constant`, an `arith.addi` for integer addition, and a `scf.yield` to return values from the loop body. This structure showcases the MLIR design to represent control flow and computation through nested regions and blocks, enabling flexible and extensible intermediate representations for the compiler infrastructure.

Figure 1. Visual hierarchical structure of the Example 9

2.3. Passes

MLIR, like LLVM, offers a wide range of **passes** that can be applied in the IR to optimize it and transform it in different ways. Some of the most useful optimization passes are:

- `cse`: This pass applies common sub-expression elimination and dead code elimination. This is a very useful pass, since it can eliminate that is produced by other passes.
- `inline`: Pass which inlines function calls.

- `one-shot-bufferize`: Pass from the `bufferization` dialect which bufferizes operations by converting from tensor semantics to memref semantics. This means assigning actual buffers to used tensors.
- `affine-loop-unroll`: Pass which unrolls `affine` loops. This means that it decreases the number the number of iterations in a loop by replicating its body a constant number of times.

Another useful application of passes is to convert operations from one dialect to another to move from levels of abstraction, in a process called **lowering**. These are some of the most used conversion passes:

- `convert-scf-to-cf`: Pass that lowers control flow abstractions. Effectively, it replaces structured control flow with a CFG.
- `convert-linalg-to-affine-loops`: Pass that lowers high-level linear algebra operations from the `linalg` dialect into affine loops, from the `affine` dialect.
- `convert-to-llvm`: Generic pass that tries to convert the IR into the `llvm` dialect, which can later be translated into LLVM IR. In general, it may be easier to use dialect-specific passes, such as `convert-func-to-llvm` or `convert-cf-to-llvm`.

MLIR offers a very robust infrastructure for implementing your own passes for your own dialects. Custom passes can range from simple graph rewrites implemented with pattern matching, to more complex transformations. MLIR's pass infrastructure also allows you to create your own pass pipelines that can be integrated into your compiler workflow.

3. Building with MLIR

In this section, we provide how to build the MLIR. As mentioned in Section 1, MLIR is part of the LLVM project. Therefore, its construction follows a similar construction. For Section 4, we will use the following requirements:

- LLVM $\geq$ 21 : llvm.org
- Python $\geq$ 3.10 : python.org
- CMake $\geq$ 3.20.0 : cmake.org
- Ninja $\geq$ 1.10 : ninja-build GitHub

3.1. Setting Up the MLIR Environment

The first step is to clone the LLVM project and then build the project using CMake. After that, we will use the ninja to build the project. You can use other methods to build the system. Finally, you can check if the installation is fine.

1. Clone the LLVM Project

```
git clone https://github.com/llvm/llvm-project.git
cd llvm-project
```

If you only want the latest stable version:

```
git checkout release/21.x  # or the latest stable branch
```

2. Configure the Build Create a build directory and run CMake:

```
mkdir build
cd build
cmake -G Ninja ../llvm \
   -DLLVM_ENABLE_PROJECTS="mlir" \
   -DLLVM_TARGETS_TO_BUILD="host" \
   -DLLVM_ENABLE_ASSERTIONS=ON \
   -DCMAKE_BUILD_TYPE=Release \
   -DLLVM_BUILD_EXAMPLES=ON \
   -DMLIR_BUILD_EXAMPLES=ON
```

3. Build MLIR:

```
ninja
# or if you're using Make:
make -j$(nproc)
```

4. Verify the Build (Optional)

```
ninja check-mlir
```

For more information, you can access the MLIR website.

The repository on GitHub includes a utility script to easily build the MLIR tutorial project. To use it, you must set the LLVM_BUILD_DIR variable to the directory where you have built LLVM. Here is an example:

```
export LLVM_BUILD_DIR=path/to/llvm-project/build
./scripts/build.sh
```

This will output the sblp-opt binary in the ./build/bin directory.

4. MLIR in Practice

This section gently introduces how to create MLIR passes by discussing a simple end-to-end project: to build a highly optimized matrix multiplication program. In the first part (Section 4.1), we'll develop a custom pass that tiles the input tensors by a factor equal to the Greatest Common Divisor (GCD) among the tensors' dimensions. In the second part (Section 4.2), we'll show how to build a custom pipeline to leverage multiple passes, sequentially, in a single run.

4.1. Case Study: a custom pass for matrix multiplication

We will generate a highly optimized matrix multiplication function by starting from a high-level MLIR representation of a 2D matmul operation and progressively lowering it to LLVM bytecode. This process leverages advanced tensor-level transformations and the full suite of LLVM optimization passes to ensure performance and correctness. The lowering pipeline safely transitions from abstract tensor computations to concrete implementations, converting tensors into buffers and ultimately into memory references suitable for low-level execution.

We will apply the set of passes presented in Script 10, in sequence, to an abstract representation of the general matrix multiplication algorithm (4.1). All but `gcd-tiling` are passes from the default toolkit MLIR offers *out of the box*—`gcd-tiling` is the custom pass we'll develop shortly. Most of the passes were introduced in section 2.3; the remaining are used to safely lower from abstract dialects, such as `arith`, to the LLVM dialect[2].

Script 10 *Example of calling the program.*

```
./build/bin/sblp-opt examples/matmul.mlir \
--one-shot-bufferize="bufferize-function-boundaries" \
--convert-linalg-to-loops \
--gcd-tiling \
--convert-scf-to-cf \
--convert-cf-to-llvm \
--convert-arith-to-llvm \
--convert-func-to-llvm \
--convert-index-to-llvm \
--finalize-memref-to-llvm \
--reconcile-unrealized-casts \
| mlir-translate -mlir-to-llvmir -o matmul.ll
```

We start from the code below using a multiplication between two tensors `A:<1024x256xf32>` and `B:<256x1024xf32>`, to yield the tensor `C:<1024x1024xf32>`.

```
#matmul_accesses = [
  affine_map<(m, n, k) -> (m, k)>,
  affine_map<(m, n, k) -> (k, n)>,
  affine_map<(m, n, k) -> (m, n)>
]

#matmul_trait = {
  doc = "C(m, n) += A(m, k) * B(k, n)",
  indexing_maps = #matmul_accesses,
  iterator_types = ["parallel", "parallel", "reduction"]
}

module {
  func.func @matmul(%A: tensor<1024x256xf32>, %B: tensor<256
     x1024xf32>, %C: tensor<1024x1024xf32>) {
    %result = linalg.generic #matmul_trait
      ins(%A, %B : tensor<1024x256xf32>, tensor<256x1024xf32>)
      outs(%C : tensor<1024x1024xf32>) {
      ^bb0(%a: f32, %b: f32, %c: f32):
      %prod = arith.mulf %a, %b : f32
      %sum = arith.addf %c, %prod : f32
```

[2]The LLVM dialect is NOT the same as LLVM IR. It is meant to map "LLVM IR into MLIR by defining the corresponding operations and types". You can learn more about it at https://mlir.llvm.org/docs/Dialects/LLVM/

```
      linalg.yield %sum : f32
    } -> tensor<1024x1024xf32>
    return
  }
}
```

One can summarize the process of building a custom pass as a three-step process:

1. Define an interface (Section 4.1.1).
2. Implement it (Section 4.1.2).
3. Add it to the Pass Registry (Section 4.1.3).

Each of these steps will be discussed in the subsections that follow.

4.1.1. Interface definition

The code 11[3] presents the definition of the interface of the *Greatest Common Divisor* (GCD) tiling pass interface. Lines 2-3 tells the pass inherits from `OperationPass` and runs on `FuncOps`. Lines 5-6 are the methods that we will implement in Section 4.1.2. Lines 10-12 are the pass name and decription for the `opt` tool. Finally, Line 16 is the interface with the `opt` tool, while line 17 allows the pass to be called from another pass—we'll discuss it further in Section 4.2.

Code 11 *GCD Tiling Pass Interface.*

```
class GCDTilingPass
    : public mlir::PassWrapper<GCDTilingPass,
                               mlir::OperationPass<mlir::func::
                                  FuncOp>> {
private:
  int64_t computeTilingSize(mlir::linalg::LinalgOp op);
  void runOnOperation() override;

  llvm::StringRef getArgument() const final { return "gcd-tiling
     "; }

  llvm::StringRef getDescription() const final {
    return "Tile loops by a factor of the greatest common
       divisor between "
           "tensor dimensions.";
  }
};

void registerGCDTilingPass();
std::unique_ptr<mlir::OperationPass<mlir::func::FuncOp>>
   createGCDTilingPass();
```

[3] Access the full program on GitHub

4.1.2. Pass Implementation

Our 2D pass tiles matrices using dimensions' Greatest Common Divisor (GCD) as the tiling factor. To achieve our goal, we will:

- Traverse the IR and collect operations that can be tiled following this strategy
- For each *tileable* operation,
  - Compute the dimensions' GCD
  - Emit the tiled operation
  - Replace the original operation with the tiled version

Luckily for us, MLIR offers a lot of utility out of the box!

In Code 12, we override the virtual `runOnOperation` method to traverse the intermediate representation (IR) and identify operations that are suitable for tiling. Our tileability criterion is based on whether all operands of an operation have static shapes, ensuring predictable memory access patterns and transformation safety. Once identified, these tileable operations are added to a worklist, which will be processed later to apply the desired optimizations and transformations.

Code 12 *GCD Implementation.*

```
1  void GCDTilingPass::runOnOperation() {
2    auto F = getOperation();
3    std::vector<linalg::LinalgOp> ops;
4
5    F.walk([&](linalg::LinalgOp op) {
6      bool tilable = true;
7
8      // Only tile operations that take tensors with static shapes
           for operands.
9      for (auto operand : op->getOperands()) {
10       if (auto operandType = llvm::dyn_cast<ShapedType>(operand.
             getType())) {
11         tilable &= ShapedType::isStaticShape(operandType.
               getShape());
12       } else {
13         tilable = false;
14       }
15     }
16
17     if (tilable)
18       ops.push_back(op);
19   });
```

And for each operation we have previously collected, compute the GCD-based tile size (Code 14).

Code snippet 14 computes the tile size from the dimensions for each operation we've previously collected. In lines 6, we obtain the operand type—an object with the tensor's dimensions and the *type* (eg., 32-bit floating point) of its elements. In line 7,

we obtain its shape: an `ArrayRef` object with n elements, where n is the degree of the tensor; in this case, $n = 3$. We load all of the dimensions into a single vector, `collectedDims`, in lines 8-10, and we use it to iteratively compute the GCD between all of the operands' dimensions (14-17).

Code 13 *Compute Tiling Size.*

```
int64_t GCDTilingPass::computeTilingSize(linalg::LinalgOp op) {
  std::vector<int64_t> collectedDims;

  // Collect the dimensions from all operands.
  for (auto operand : op->getOperands()) {
    auto operandType = llvm::dyn_cast<ShapedType>(operand.
        getType());
    llvm::ArrayRef<int64_t> shape = operandType.getShape();
    for (int64_t dim : shape) {
      collectedDims.push_back(dim);
    }
  }

  // Compute the greatest common divisor between dimensions.
  int64_t tileSize = collectedDims[0];
  for (size_t idx = 1; idx < collectedDims.size(); idx++) {
    tileSize = greatestCommonDivisor(tileSize, collectedDims[idx
        ]);
  }

  return tileSize;
}
```

Now, we emit the new, tiled operation and replace the untiled version with it. In Lines 9-10, we define the tile size, per dimension, and the type of loop to tile. We safely generate the tiled version of the operation in lines 12-16. Finally (18-21), we replace the untiled operation uses with the tiled one before deleting it.

Code 14 *Compute Tiling Size.*

```
  ...
  IRRewriter rewriter(F.getContext());

  for (auto op : ops) {
    linalg::LinalgTilingOptions options;

    auto tileSize = computeTilingSize(op);

    options.setTileSizes({tileSize, tileSize});
    options.setLoopType(linalg::LinalgTilingLoopType::Loops);

    auto tiledOp = tileLinalgOp(rewriter, op, options);

    if (failed(tiledOp)) {
```

```
15        break;
16      }
17
18      for (size_t idx = 0; idx < op->getNumResults(); idx++) {
19        op->getResults()[idx].replaceAllUsesWith(tiledOp->
              tensorResults[idx]);
20      }
21      op->erase();
22    }
23  }
```

4.1.3. Pass registration

The final step is to register the pass in the Pass Registry. The Pass Registry is MLIR's infrastructure to manage available passes, exposing them so user's can invoke them from the CLI interface—i.e., through the `opt` tool (`sblp-opt`, in this case). Pass registration is straightforward: we simply invoke `PassRegistration` while passing the pass class as the template (Code 15), then we make it available in our `opt` tool (Code 16, line 7). Additionally, we implement the `createGCDTilingPass` so this pass can be called from another pass—that is, to be part of a pipeline; this topic is discussed in further detail in the next section.

Code 15 *Pass registration.*

```
1  void registerGCDTilingPass() { PassRegistration<GCDTilingPass>()
       ; }
2
3  std::unique_ptr<OperationPass<func::FuncOp>> createGCDTilingPass
       () {
4    return std::make_unique<GCDTilingPass>();
5  }
```

Code 16 *Main Program.*

```
1  int main(int argc, char **argv) {
2    mlir::registerAllPasses();
3
4    sblp::registerStrengthReductionPass();
5    sblp::registerGCDTilingPass();
6
7    mlir::DialectRegistry registry;
8    mlir::registerAllDialects(registry);
9
10   return failed(mlir::MlirOptMain(argc, argv, "SBLP 2025 MLIR
         Tutorial test\n",
11                                   registry));
12 }
```

4.2. Pass Pipeline

As projects grows more complex, the amount of passes it uses will also grow. As such, invoking passes as we've shown Script 10 becomes cumbersome. Fortunately, one can implement a pass pipeline: a pass that is responsible for orchestrating calls of other passes—the previously mentioned script becomes way simpler, as in Script 17. In this section, we'll discuss how to build such pipeline.

Script 17 *Refactoring the program call with pass pipelines.*

```
./build/bin/sblp-opt examples/matmul.mlir \
--optimize-loops \
--lower-to-llvm-dialect \
-o matmul.llvm.mlir
```

To create a pass pipeline *is itself* to create a pass. So, we'll follow the same steps as those we've discussed in subsection 4.1.2. The main difference is that we won't need a new class—as our pipeline passes will just add sequences of passes, something as simple as a pass registration function will do. Code snippet 18 displays the signatures of the two passes we'll build: `LoopOptimizationPipeline`, that makes all of the loop-related transformations, and `LowerToLLVMPipeline`, that will sequentially lower our IR to the LLVM dialect.

Code 18 *Pass Pipeline header.*

```
namespace sblp {

void registerLoopOptimizationPipeline();
void registerLowerToLLVMPipeline();

} // namespace sblp
```

In the next step, we'll implement these pass registration functions. Such functions receive a `OpPassManager` object as parameter, and add passes to it in the same order we want them to run—MLIR will automatically chain the output of pass n as the input of pass $n + 1$. While MLIR may parallelize pass executions—for instance, to optimize two functions at the same time—, it is guaranteed that the each instance will run the passes in the same order they were added to the `OpPassManager` object.

Code snippets 19 and 20 display the implementation of the previously introduced pass pipelines. It is worth noting that a pass added to the `OpPassManager` object can be parameterized: lines 3-6 of 19 will enable the bufferization of function boundaries as the `OneShotBufferizePass` is added to the pipeline.

Code 19 *LoopOptimizationPipeline implementation.*

```
void addLoopOptimizationPipeline(OpPassManager &pm) {
  // Bufferization (tensor->memref)
  mlir::bufferization::OneShotBufferizePassOptions
     bufferizationOptions;
  bufferizationOptions.bufferizeFunctionBoundaries = true;
```

```
  pm.addPass(
      mlir::bufferization::createOneShotBufferizePass(
          bufferizationOptions));

  // Apply tiling and lower linalg to scf loops
  pm.addPass(sblp::createGCDTilingPass());
  pm.addPass(createConvertLinalgToLoopsPass());
}
```

Code 20 *LowerToLLVMPipeline implementation.*

```
void addLowertoLLVMPipeline(OpPassManager &pm) {
  // Lower structured control flow
  pm.addPass(createSCFToControlFlowPass());
  pm.addPass(createConvertControlFlowToLLVMPass());

  // Lower memref ops that use strided metadata
  pm.addPass(memref::createExpandStridedMetadataPass());
  pm.addPass(createLowerAffinePass());

  // Finalize llvm conversion
  pm.addPass(createArithToLLVMConversionPass());
  pm.addPass(createConvertFuncToLLVMPass());
  pm.addPass(createConvertIndexToLLVMPass());
  pm.addPass(createFinalizeMemRefToLLVMConversionPass());
  pm.addPass(createReconcileUnrealizedCastsPass());
}
```

Finally, we register the passes so they can be invoked from the CLI tool. First, we'll implement the pass registration functions, as in Code 21; and then register it in the `opt` tool as shown in Code 22.

Code 21 *Pass pipelines registration.*

```
namespace sblp {

void registerLoopOptimizationPipeline() {
  PassPipelineRegistration<>("optimize-loops", "Bufferize and
      tile loops.",
                             addLoopOptimizationPipeline);
}

void registerLowerToLLVMPipeline() {
  PassPipelineRegistration<>("lower-to-llvm-dialect",
                             "Lower IR to the `llvm` dialect.",
                             addLowertoLLVMPipeline);
}

} // namespace sblp
```

Code 22 *Pass pipeline registration in the* `opt` *tool.*

```
...
sblp::registerLoopOptimizationPipeline();
sblp::registerLowerToLLVMPipeline();
...
```

5. Conclusion

This paper serves as a brief and gentle introduction to the Multi-level Intermediate Representation framework—MLIR. It has two main parts. Part 1 serves as an introduction that motivates why MLIR was conceived in the first place and overviews its architecture and main concepts (Sections 1 and 2, respectively). Part 2 is a practical example: after building the project (Section 3), we've presented a hands-on example of a custom pass to produce a highly optimized matrix multiplication program (Section 4).

Our goal with this paper is to offer an introduction to MLIR that is lighter and easier to follow than the official MLIR tutorial. As such, this paper does not cover all of the MLIR fundamentals—we've deliberately chosen not to cover TableGen or debugging, for instance. Given that, we highly recommend those who are interested to browse the official MLIR documentation and to interact with the community for further learning.

References

[1] Tobias Gysi, Christoph Mueller, Oleksandr Zinenko, Stephan Andreas Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. Domain-specific multi-level rewriting for hpc: A case study with mlir. *ACM Transactions on Architecture and Code Optimization*, 4:51:1–51:23, 2021.

[2] Abhinav Kumar, Atharva Khedkar, Hwisoo So, Megan Kuo, Ameya Gurjar, Partha Biswas, and Aviral Shrivastava. Dsp-mlir: A domain-specific language and mlir dialect for digital signal processing. In *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '25, page 146–157. ACM, June 2025.

[3] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[4] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[5] Dewei Wang, Wei Zhu, Liyang Ling, Ettore Tiotto, Quintin Wang, Whitney Tsang, Julian Opperman, and Jacky Deng. Ml-triton, a multi-level compilation and language extension to triton gpu programming, 2025.