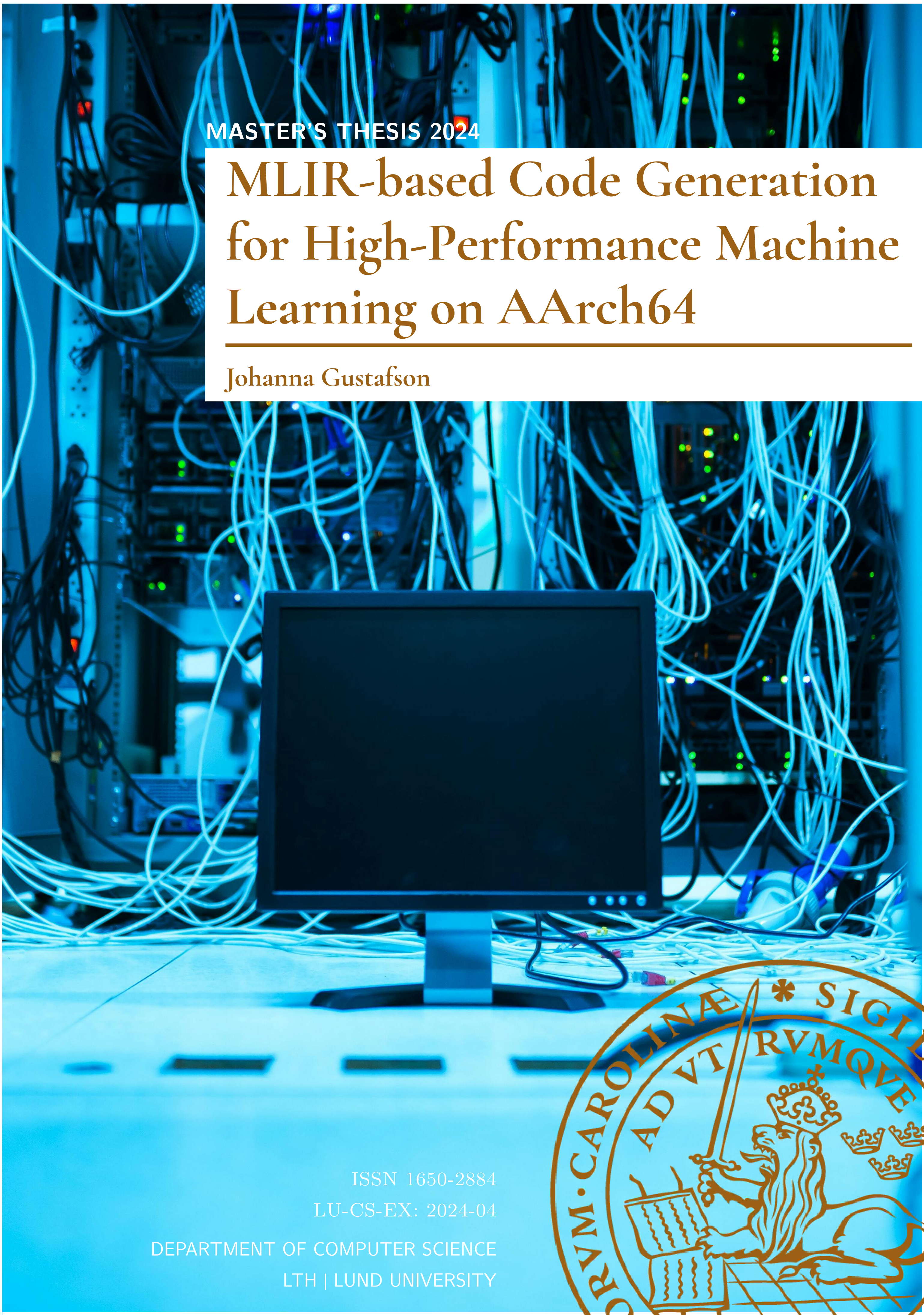
MASTER'S THESIS 2024

MLIR-based Code Generation for High-Performance Machine Learning on AArch64

Johanna Gustafson



ISSN 1650-2884
LU-CS-EX: 2024-04

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-04

MLIR-based Code Generation for High-Performance Machine Learning on AArch64

MLIR-baserad kodgenerering för högpresterande maskininlärning på AArch64

Johanna Gustafson

MLIR-based Code Generation for High-Performance Machine Learning on AArch64

(Building a retargetable high-performance DGEMM routine with an MLIR-based approach)

Johanna Gustafson
`jo2718gu-s@student.lu.se`

January 22, 2024

Master's thesis work carried out at Arm Sweden AB.

Supervisors: Jonas Skeppstedt, `Jonas.Skeppstedt@cs.lth.se`
Fredrik Knutsson, `Fredrik.Knutsson@arm.com`
Per Åstrand, `Per.Astrand@arm.com`

Examiner: Flavius Gruian, `Flavius.Gruian@cs.lth.se`

Abstract

Given the growing complexity of machine learning architectures driving the state-of-the-art, inference acceleration is an important aspect to consider in the development of machine learning systems. Machine learning frameworks often address this challenge using hand-crafted compute libraries optimized for a narrow range of hardware devices — often implemented by the hardware vendors themselves — meaning significant engineering effort is needed to extend to new platforms. For that reason, we highlight a modular approach in the design of machine learning systems, utilizing the MLIR (Multi-Level Intermediate Representation) compiler framework of the LLVM project. This notion is strengthened through a demonstration of an MLIR-based implementation of an optimized double-precision GEMM (GEneral Matrix Multiply) routine, which is shown to reach a performance of 86% of the theoretical machine peak on a single Neoverse-N1 core. Given the reusable and extensible nature of MLIR, we believe that there is value in developing MLIR-based compilers for machine learning software.

Keywords: MLIR, GEMM, machine learning systems, optimizing compilers, AArch64

Acknowledgements

I would like to acknowledge my supervisors, Jonas Skeppstedt from the Department of Computer Science at LTH, and Fredrik Knutsson and Per Åstrand from Arm in Lund, for their contributions to this master's thesis, in helping to lay out the aim of the project and providing experience and knowledge relevant to understanding MLIR and computer architecture. I would also like to thank Thomas Preud'homme from Arm in Cambridge who was happy to share his engineering expertise during my time as an intern, and Flavius Gruian from the Department of Computer Science at LTH for taking on the role as examiner.

Contents

Chapter 1

Introduction

With the advancement of machine learning, especially artificial neural networks, the complexity of the underlying architectures driving the state-of-the-art is increasing. This is exemplified by the deep learning model GPT-4, which supposedly withholds around 1.8 trillion parameters across 120 layers [52, 17], making it more than 10 times larger than its predecessor GPT-3 [52]. The rivaling Gemini AI of Google is said to have upwards 60 trillion parameters [17]. Consequently, machine learning inference is an important factor to consider beside the pure capabilities of machine learning models. The problem of inference acceleration can be divided into three main categories; hardware, software and algorithmic [16]. On an algorithmic level, the idea is to utilize methods that essentially remove or substitute components of the model itself [16], reducing complexity without affecting the capabilities of the model. On the other end of the inference acceleration stack, the idea is to make use of advanced computation units in the hardware [16]. Although, most relevant to this thesis is the software aspect — methods where software is used to accelerate inference without changing the architecture [16]. Software accelerators are further divided into two subcategories; low-level libraries and graph compilers [16]. These differ in that the latter utilizes optimized code generation to form a bridge between the machine learning model and target hardware, while the former provides highly-tuned implementations for standard routines within machine learning such as forward and backward convolution, pooling, normalization, and activation layers [16].

The way in which the inference problem is handled differs between different machine learning frameworks. For TensorFlow, one of the most popular machine learning frameworks among data scientists [22], Intel oneAPI Deep Neural Network Library (OneDNN) is turned on by default in all Linux x86 packages, delivering up to three times improved performance by accelerating key performance-intensive operations, including matrix multiplication [13]. Though being an Intel library, experimental support is included for AArch64, enabling Arm Compute Library (ACL) for machine learning applications and Arm Performance Libraries (ArmPL) for general matrix multiply operations, respectively [14]. As such, TensorFlow employs ACL to accelerate performance on AArch64 CPUs [58]. ACL is a collection of low-level machine learning functions optimized for Cortex-A CPU, Neoverse and Mali GPU architectures [26], while ArmPL provides optimized standard core math libraries for numerical applications on 64-bit Arm based processors [27].

OneDNN, ArmPL, and ACL are just some of the libraries that implement routines accelerating the performance of machine learning applications. These routines have been developed and carefully hand-optimized with significant engineering effort, often by the hardware vendors themselves. Thus, an exceptional knowledge of the hardware and low-level code optimization is imperative to produce near-peak performance, making the process of deploying machine learning workloads to new hardware platforms costly and tedious.

The MLIR (Multi-level Intermediate Representation) project addresses the challenge of expanding software to heterogeneous hardware platforms by introducing a reusable and extensible framework for compiler development. MLIR was first announced by TensorFlow, Google, in the EuroLLVM conference in April of 2019 [57], but was later contributed to the LLVM Foundation in September of that year [23]. The design of MLIR is a hybrid of notions from traditional three-address SSA representations and polyhedral loop optimization works, optimized to represent, analyze, and transform high-level dataflow graphs as well as target-specific code generated for high-performance data-parallel systems [39]. MLIR's single continuous design provides a framework to lower from arbitrary dataflow graphs to high-performance target-specific code for a wide range of parallel architectures, enabled through MLIR's strong representational capabilities [39].

By functioning as a foundation for various compiler systems with the ability of targeting heterogeneous hardware, the MLIR framework has the ambition to combat software fragmentation and reduce the cost of building domain-specific compilers [40].

1.1 Related work

Nicolas Vasilache et. al. made a case for MLIR in their article titled *Composable and Modular Code Generation in MLIR* [62], published in 2022. The authors emphasized that machine learning systems, runtimes and compilers do not compose properly, and proposed an MLIR-based design aiming at providing unprecedented degrees of modularity, composability and genericity [62]. A number of single-thread CPU experiments proved that kernels of high relevance to the machine learning community implemented with an MLIR-based approach were able to reach near-peak performance [62].

Many novel compilers are built on MLIR technology, including Accera [link], Catalyst [link], Firefly [link], PlaidML [link] and Pylir [link] [43]. One of the most prominent projects to utilize MLIR is OpenXLA; an open-source machine learning compiler ecosystem made available to the public in March of 2023, including the XLA, IREE, and StableHLO projects — all of which leverage MLIR [50]. XLA (Accelerated Linear Algebra) is a compiler for machine learning software that supports several machine learning frameworks, including TensorFlow, PyTorch and JAX [59]. IREE, similar to XLA, is an end-to-end compiler and runtime that lowers machine learning models to a unified IR and is supported by a variety of machine learning frameworks, including the aforementioned three [21]. StableHLO, on the other hand, is an operation set for high-level operations (HLO) that functions as a portability layer between machine learning frameworks and compilers [50].

Before the announcement of MLIR, Glow [49] and TVM [6] have been proposed as compilers for neural networks and deep learning, respectively, both tackling the challenge of generating highly optimized code for heterogeneous hardware [49, 6].

1.1.1 An Early Case Study with GEMM

HIGH PERFORMANCE CODE GENERATION IN MLIR: AN EARLY CASE STUDY WITH GEMM is an article published in 2020 [5], analyzing various implementations of equation 1.1, shown below.

$$C := C + AB \tag{1.1}$$

This operation entails adding the result of the matrix multiplication $\boldsymbol{AB}$ to matrix $\boldsymbol{C}$, and assigning $\boldsymbol{C}$ with the sum. This article is given its own section as it has been a big influence to this thesis and our approach.

The author, Uday Bondhugula, begins by presenting a performance-baseline by benchmarking various 2088×2048 double-precision implementations of equation 1.1: naive-nest in C compiled with GCC and Clang, followed up with the hand-crafted matrix multiplication routines of the OpenBLAS, BLIS and MKL (Math Kernel Library — a library of math functions for Intel CPUs and GPUs [15]) libraries. As one can expect, the libraries reach performances of 85% and 92% of the theoretical machine peak [5], being 75.2 GFLOPS (10^9 Floating-Point Operations Per Second) per core on an Intel-based system, while GCC and Clang stay at 0.6% and 6% respectively [5]. The implication is that OpenBLAS, BLIS and MKL have the advantage of being specialized to efficiently run linear algebra problems or machine learning workloads, while general-purpose compilers such as GCC and Clang are not designed to optimize appropriately.

In the remainder of the article, Bondhugula implements equation 1.1 as an operation in MLIR and iteratively introduces optimization techniques, comparing the performance (measured in FLOPS) against the baseline values presented in the beginning. For the purposes of the article, Bondhugula created a `-hopt` pass which expands the high-level operation `hop.matmul` into MLIR-native mid-level code that performs a matrix multiplication of specified matrices. The syntax is [5]

```
hop.matmul %A, %B, %C {some_attribute = 2, some_other_attribute = 4} : (memref<2088
    x2048xf64>, memref<2048x2048xf64>, memref<2088x2048xf64>)
```

where `%A` and `%B` correspond to the source operands and `%C` the destination operand of equation 1.1, and `some_attribute` and `some_other_attribute` represent attributes of the operation. When compiling with the `-hopt` pass, `hop.matmul` expands to a naive 3-d loop nest based on the `affine` dialect, corresponding to the previously implemented naive-nest in C. As such, the optimization strategies discussed in the article are examples of, or work in conjunction with, loop transformations. The optimization techniques are iteratively applied and discussed in order of: tiling, explicit copying or packing, unroll and jam, and vectorization, all made available through `-hopt`. Lowering the MLIR to LLVM IR with no optimization directives (beside `-O3` in the LLVM pipeline) lead to a performance in line with that of code compiled with `clang -O3` — quite poor. However, it must be noted that this is because both processes fundamentally are the same; the input code is translated from respective source language to LLVM IR and then go through the same LLVM -O3 pipeline [5]. Since the MLIR project is not a compiler in itself — although providing tools to lower and execute MLIR source code — there is essentially no point in using MLIR unless one builds upon it. Bondhugula finds that when implementing all aforementioned optimizations, optimally configured, MLIR performs at around 82.4% of the theoretical peak [5], concluding that MLIR-based code generation is on-par or close to what was achieved with expert-written assembly.

Given this promising result, we hypothesize that we can draw a similar conclusion; we will be able to reach the performance of representative high-performance compute libraries using MLIR-based code generation, although on an Arm-based setup.

1.2 Research Objective

On a high level, this master's thesis aims to investigate the potential, in terms of performance and composability, of MLIR-based code generation against high-performance compute libraries commonly used in or for machine learning on Arm-based CPUs (Central Processing Units), specifically the AWS Graviton2 CPU with Neoverse-N1 cores. Due to the complexity of machine learning models driving the state-of-the-art and the width of the field of machine learning, we are limiting our analysis to one fundamental computation — matrix multiplication. In terms of high-performance compute libraries, ACL would be a natural choice given that it implements machine learning functions optimized for Neoverse architectures, but given that we are going to specifically analyze matrix multiplication, we have determined that ArmPL is a suitable alternative. In addition to ArmPL, we are also including the BLAS-like Instantiation Software (BLIS) framework, proposed by Field G. Van Zee and Robert A. van de Geijn at The University of Texas at Austin in 2015 [60], due to the availability of in-depth documentation of implemented kernels.

1.2.1 Research Questions

The following research questions detail the aim of our research.

RQ1. To what extent do the optimization techniques loop tiling, explicit copying, loop unrolling, scalar replacement and vectorization improve the performance of GEMM (GEneral Matrix Multiply), and how are they implemented in MLIR?

RQ2. Is MLIR-based code generation able to target the vector instructions of Arm Neon (SIMD (Single Instruction Multiple Data) architecture extension)?

RQ3. What are the tradeoffs between using MLIR and high-performance compute libraries, specifically ArmPL and BLIS, in building and employing routines for high-performance machine learning, in terms of performance and composability?

The first two questions, RQ1 and RQ2, directly follow from the experiments we are conducting in this thesis. Furthermore, the outcome of our experimentation will be used to answer the third question, RQ3.

1.2.2 Contribution to Research

Given the scale of machine learning applications in use today, there is reason to introduce and further research novel compilers and infrastructures for machine learning, for improved efficiency across a wide range of use-cases. This master's thesis will ideally contribute to development of knowledge by presenting the MLIR compiler infrastructure as a potential solution to issues within machine learning, strengthening the claims of previous works such as that of Vasilache et. al. in 2022 [62]. Given the novelty of MLIR and its descendants, we also contribute by shedding light on its existence.

Though there already exist articles discussing the performance of MLIR-based code generation, there is little diversity in what hardware architecture is targeted, and MLIR is continuously developing. This master's thesis draws a lot of inspiration from the works of Nicolas Vasilache et. al. [62] and especially Uday Bondhugula [5], but is set apart in that an Arm-based rather than Intel-based setup is used and a later version of MLIR is investigated.

Chapter 2
Background

2.1 Compiler Architecture

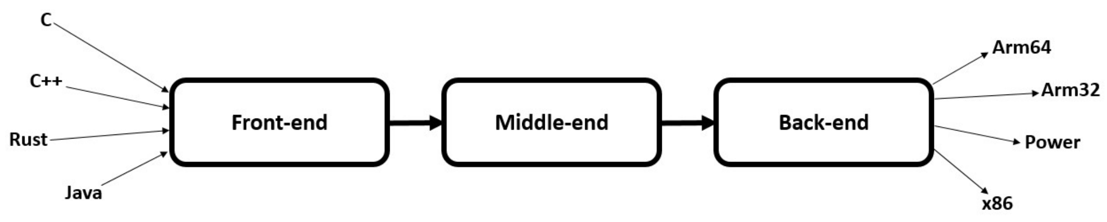


Figure 2.1: Three-stage compiler structure.

A compiler is a computer program that translates input code from one programming language (source) to another (target) [1]. The compilation process is summarized in figure 2.1, where C, C++, Rust and Java on the left-hand side are example source programming languages and Arm64, Arm32, Power and x86 on the right-hand side are example targets. It should be mentioned that these example targets are actually instruction set architecture *families*, or subsets of them. The target for a back end would rather be the implementation of the ISA (Instruction Set Architecture), such as a CPU. Beside compilers that translate high-level code to low-level code, as illustrated in figure 2.1, there exist many types of compilers for essentially any combination of abstraction level and direction. But, for the purposes of this thesis, we are going to focus on the former and CPU code generation in particular.

A compiler may implement all or some of the following phases;

- pre-processing,
- lexical analysis (lexer),
- syntax analysis (parser),

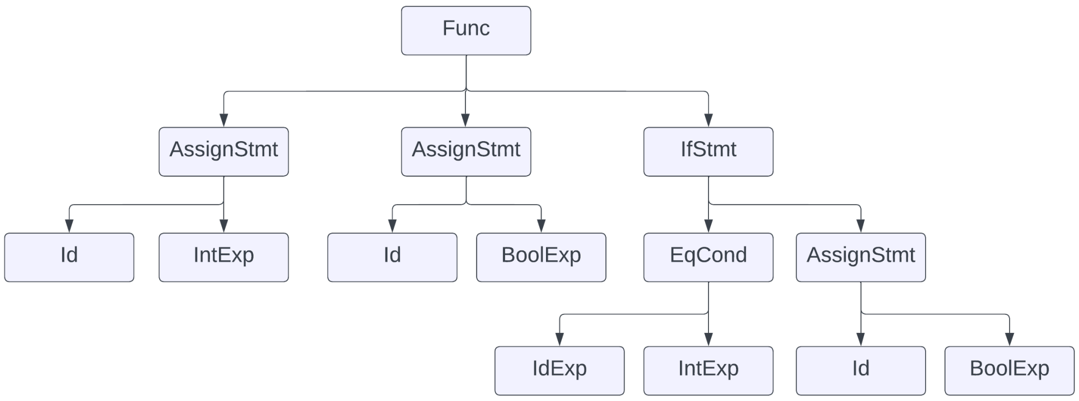


Figure 2.2: An abstract syntax tree representation of the source code presented in listing 2.1.

- semantic analysis,
- intermediate code generation,
- machine-independent code optimization,
- target code generation, and finally
- machine-dependent code optimization,

in the above or similar order [1]. Furthermore, the first five phases can be considered to be a part of the front end, the last two of the back end and the remaining one, machine-independent code optimization, of the middle end.

In the initial phases, the source code is broken down to a sequence of *tokens*, containing information for each meaningful sequence of characters [1]. These tokens are then structured according to the formal grammar of the programming language, represented by a *syntax tree* [12, 1]. The compiler may produce an *abstract syntax tree* (often abbreviated to AST) [12], where each node represents an operator, and the children nodes are its operands [1]. Figure 2.2 illustrates a sample AST for the source code in listing 2.1.

Listing 2.1: A sample program written in a made-up programming language. Note that the formal grammar of this language is highly simplified for demonstrative purposes.

```
function main():
    a = 1;
    b = false;
    if (a == 1):
        b = true;
```

The compiler may construct one or more intermediate representations, IRs, in the translation process. The abstract syntax tree, as seen in figure 2.2, is an example of a graphical IR [12, 1]. Before entering the back end of the compiler, it is typical to generate a low-level or machine-like IR that is easy to translate into the target machine [1]. The IR generation phase does not necessarily complete in a single step, but may consist of multiple internal levels.

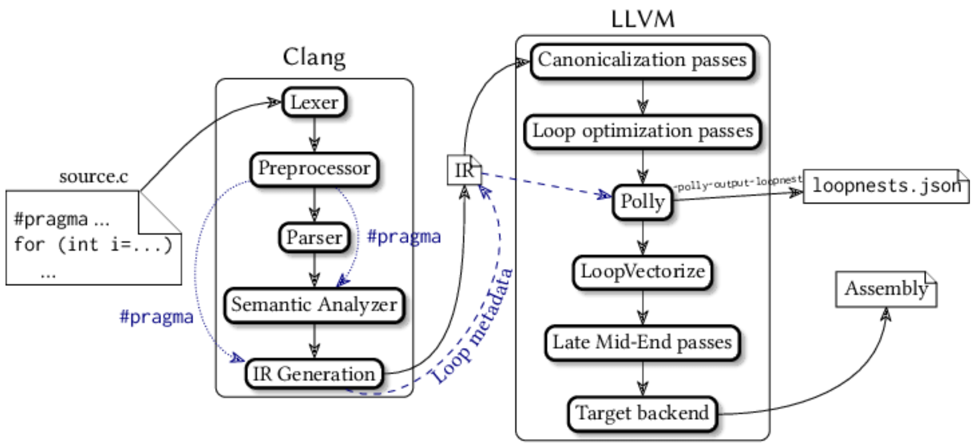


Figure 2.3: Clang/LLVM compiler architecture. By Jaehoon Koo et. al. [link], licensed under Creative Commons Attribution 4.0 International.

Steven Muchnick, for example, describes an advanced compiler design with three IR levels; one high-level, one medium-level and one low-level [45].

The middle end performs machine-independent code optimizations, e.g., *dead code elimination*; eliminating redundant or useless code, and *constant propagation*; substituting the values of known constants in expressions, on the IR. The back end performs target code generation and, in the process, register allocation. As implied by the listed phases of a compiler, the generated machine code may go through another round of code optimization before being output by the compiler.

This three-stage structure of compilers allows for combining front ends for different programming languages with back ends for different target CPUs, facilitated by a shared IR and/or middle end [12]. Clang, for example, provides front ends for C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript which emit LLVM IR, and thus share the same middle- and back ends for code optimizations and target code generation [29]. Figure 2.3 illustrates the architecture of a C-compiler consisting of Clang's front end and a middle- and back end from LLVM. In the Clang section we see the five forementioned front end phases, while in the LLVM section different code optimization subphases are listed but the back end is left arbitrary — *Assembly* refers to any low-level programming language with instructions closely resembling the instructions of the machine. This is because LLVM provides back ends for many popular CPUs as well as lesser-known ones, all built around LLVM IR [31]. Beside Clang, there are many other technologies, external to the LLVM project, that emit LLVM IR. One such example is the Rust compiler, rustc [51].

GNU Compiler Collection (GCC), similar to Clang, provides front ends for multiple programming languages, including C, C++, Objective C, Fortran, Ada, Go, and D [56]. The main interface between a GCC front end and the rest of the compiler is via the IR *GENERIC* [47], as illustrated in figure 2.4. The following IR, *GIMPLE*, is a subset of GENERIC used for optimizations [47]. RTL (Register Transfer Language), unlike the C-like GENERIC and GIMPLE representations, is an assembler language for an abstract machine with infinite registers and is ideally suited for low-level transformations such as register allocation and scheduling [47]. Furthermore, a *machine description* file defines hardware-specific features and essentially adapts the back end to the target [47].

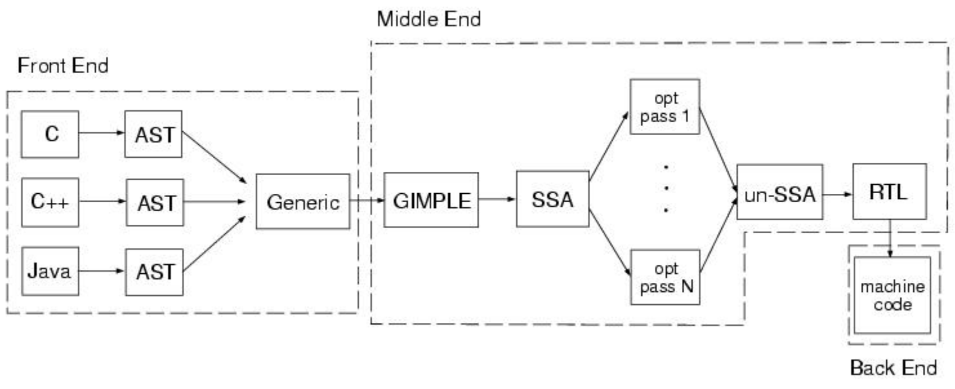


Figure 2.4: GCC architecture. By AlexeySmirnov at English Wikipedia, licensed under Creative Commons Attribution-ShareAlike 3.0 Unported.

2.1.1 Emitting LLVM IR with Clang

Using Clang's `-emit-llvm` flag in conjuction with the `-S` flag we can observe a sample LLVM IR output in assembly format. Listing C.2 in the appendix presents a sample output file from running the command on a simple function multiplying `a` with `b`, and then returning `c` wherein the result has been stored (see listing C.1 in the appendix). The output file contains a lot of information, such as the target architecture, x86-64[1], and operating system, Ubuntu, but it is the excerpt presented in listing 2.2 that directly corresponds to our multiplication function.

Listing 2.2: Excerpt from listing C.2 in the appendix.

```
define dso_local i32 @multiply() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 4, i32* %1, align 4
  store i32 10, i32* %2, align 4
  %4 = load i32, i32* %1, align 4
  %5 = load i32, i32* %2, align 4
  %6 = mul nsw i32 %4, %5
  store i32 %6, i32* %3, align 4
  %7 = load i32, i32* %3, align 4
  ret i32 %7
}
```

First, space is allocated at addresses `%1`-`%3` for variables corresponding to `a`, `b` and `c`. Then the values 4 and 10 are stored at `%1` and `%2`, respectively, and later loaded into `%4` and `%5`. These are multiplied and the result is stored into `%3` through the temporary variable `%6`, loaded into `%7` and finally returned. Clearly, this is a lower-level representation of the source function, but it is still target-independent given that there is no register allocation and the instructions are generic. Note that this IR is unoptimized. If we instead want to view an optimized output from the middle end, we also flag with `-O3`. The optimized IR has been dramatically reduced in size, as seen in listing 2.3.

[1]The command was executed on an Intel-based setup.

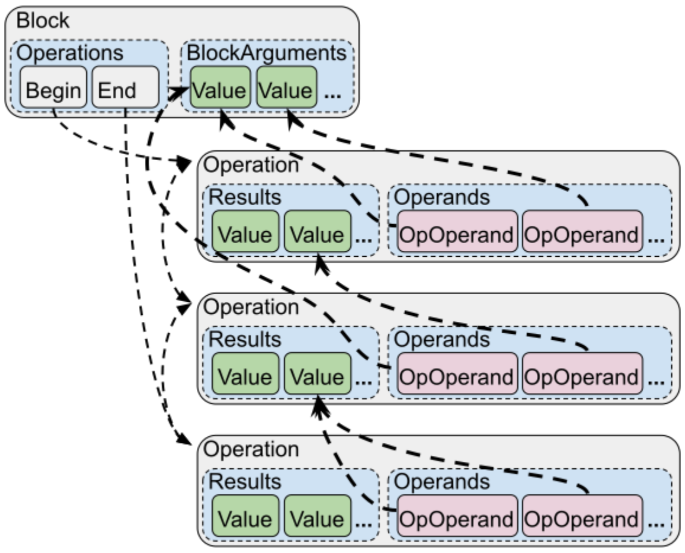


Figure 2.5: A generic example of an MLIR structure. Image taken from MLIR's *Understanding the IR structure* tutorial [link].

Listing 2.3: Output LLVM IR from running `clang -S -emit-llvm` on the multiplication function in listing C.1 in the appendix.

```
define dso_local i32 @multiply() local_unnamed_addr #0 {
  ret i32 40
}
```

This may have been achieved through a combination of constant propagation and dead code elimination, the forementioned target-independent optimizations.

2.2 The MLIR Compiler Framework

MLIR is another subproject of the LLVM Project and thus shares some aspects of Clang and other LLVM-based compiler architectures, but it differs in that it does not provide a complete front end for any programming language. MLIR is rather a framework for compiler development.

MLIR is a graphical IR consisting of nodes, called *Operations*, and edges, called *Values* [38]. The Operations are contained within Blocks and Blocks are contained within Regions [38]. Operations may also contain Regions, allowing for recursive structures [38]. Operations are the core unit of abstraction and computation [41]; beside higher-level concepts like function definitions and calls, Operations can represent lower-level concepts like target-specific machine instructions [38]. Figure 2.5 illustrates a generic example of the described hierarchical structure of MLIR, where the root of this structure is a Block containing three Operations, accessed through the Begin and End pointers. Each Operation has an arbitrary number of Results, containing Values, and Operands, containing OpOperands. The thicker arrows connects the use of a Value with its definition.

Note that there are two classes within MLIR's C++ source code which relate to operations; `Operation` and `Op` and any future mention of "operation" in the context of MLIR could refer to either. The `Operation` class is used to generally model all operations and provides a general API into an operation instance, while each type of operation is represented by an `Op`-derived class acting as a smart pointer wrapper around a `Operation *`, providing operation-specific accessor methods and type-safe properties of the operation [41].

Furthermore, MLIR's *dialects* serve as a logical grouping mechanism for operations, attributes and types [24]. The dialect namespace appears as a dot-separated prefix, e.g. `toy.constant` represents the `constant` operation of the made-up dialect `toy`. Given the representational capabilities of MLIR and its operations, a dialect can range from high-level and hardware independent to low-level and hardware aware. The `arm_neon` dialect [link] in the MLIR repository is an example of the latter, as it consists of operations that directly translate to some of the LLVM instructions targeting Arm Neon intrinsics [61]. Utilities, being operations, attributes and types, from multiple dialects may co-exist at each level of the IR [24].

At the most fundamental level, MLIR can be completely described by its name; it is an IR with multiple levels. One of the core principles of MLIR — as implied by first half of its name — is to support *progressive lowering*, meaning to convert higher-level code down to low-level representation in multiple small abstraction levels [24]. This is facilitated by MLIR's pass infrastructure, a familiar concept from LLVM [32], representing MLIR's infrastructure for transformations and optimizations. All compiler passes in MLIR inherit from the `OperationPass` class and must adhere to a set of instructions, as it otherwise could lead to problematic behavior in multithreaded and other advanced scenarios [42]. For example, a compiler pass "must not modify any state referenced or relied upon outside the current operation being operated on" ([42]). Similarly, it also "must not modify the state of another operation not nested within the current operation being operated on" ([42]) and "must not inspect the state of sibling operations" ([42]). Essentially, a compiler pass operates on a certain type of operation, independently of surrounding information.

2.2.1 Lowering MLIR to LLVM IR

In a terminal, a compiler pass is represented by its class name in lower-case with each word prefixed by a dash. Any compiler pass named `-convert-*-to-llvm` serves the purpose of converting utility of one dialect into corresponding utility in the `llvm` dialect [link]. The `llvm` dialect is an important dialect in the MLIR repository as it defines operations and types corresponding to LLVM IR, and thus lays a foundation for a smooth transition into the LLVM middle- and back ends. The following MLIR is semantically equivalent to the previously implemented multiplication function in C (see listing C.1 in the appendix).

Listing 2.4: The multiplication function in MLIR.

```
module {
  func.func @multiply() -> i32 {
    %a = arith.constant 4 : i32
    %b = arith.constant 10 : i32
    %c = arith.muli %a, %b : i32
    return %c : i32
  }
}
```

Note that `module` represents a top-level container operation, belonging to the `builtin` dialect [link]. Using MLIR Playground[2] [link]], we apply the `-convert-arith-to-llvm` pass

[2]MLIR Playground is a tool developed by MLIR-China, an independent user group based in China,

which produces the MLIR in listing 2.5,

Listing 2.5: `@multiply` with the `-convert-arith-to-llvm` pass applied.

```
module {
  func.func @multiply() -> i32 {
    %0 = llvm.mlir.constant(4 : i32) : i32
    %1 = llvm.mlir.constant(10 : i32) : i32
    %2 = llvm.mlir.constant(40 : i32) : i32
    return %2 : i32
  }
}
```

then the `-convert-func-to-llvm` which produces the MLIR in listing 2.6,

Listing 2.6: `@multiply` with the `-convert-arith-to-llvm` and `-convert-func-to-llvm` passes applied.

```
module attributes {llvm.data_layout = ""} {
  llvm.func @multiply() -> i32 {
    %0 = llvm.mlir.constant(4 : i32) : i32
    %1 = llvm.mlir.constant(10 : i32) : i32
    %2 = llvm.mlir.constant(40 : i32) : i32
    llvm.return %2 : i32
  }
}
```

and finally the `-canonicalize` pass which produces the MLIR in listing 2.7,

Listing 2.7: `@multiply` with the `-convert-arith-to-llvm`, `-convert-func-to-llvm` and `-canonicalize` passes applied.

```
module attributes {llvm.data_layout = ""} {
  llvm.func @main() -> i32 {
    %0 = llvm.mlir.constant(40 : i32) : i32
    llvm.return %0 : i32
  }
}
```

and retrieve an IR similar to the optimized LLVM IR presented in listing 2.3. The `llvm` dialect can then be exported to LLVM IR using resources from the MLIR framework. Chapter 6 of MLIR's *Toy Tutorial* [34] gives an example of how this is implemented in an MLIR-based compiler (see listing B.1 in the appendix).

This is to demonstrate how MLIR may function as an intermediate representation in a compiler. This also elaborates on the principle of progressive lowering, as we have synthetically lowered the source code to the `llvm` dialect in three levels. Note however that we have only demonstrated lowering and canonicalization and not any direct mechanism for optimization in MLIR, which compiler passes also are responsible for. This will first be introduced in section 4.1.

which allows for experimenting with MLIR without the need for installing any dependencies or setting up a build system. The system was upgraded to LLVM 16 on May 25 2023 (see commit with hash 838b6057578be921601ead56421131dc85d6c3a6 [link]), but has not been updated since.

2.3 Setup

2.3.1 Hardware

We are going to be remotely connected to an instance from the Amazon Elastic Compute Cloud (EC2). There are different instance types optimized to fit different use cases [54]. We are going to use an instance from the *Compute Optimized* category, as it is well suited for high performance computing and machine learning inference [54]. The *c6g*, *c7g* and *c7gn* instances from this category are all powered by Arm-based CPUs [54].

More specifically, we will be connected to a *c6g.metal*[3] instance, powered by Arm-based AWS Graviton2 processors [53]. The Graviton2 CPU has 64 Neoverse-N1 cores running at 2.5 GHz, with ARMv8.2-a ISA (Instruction Set Architecture) including 2×128-bit Neon, and the additional features fp16, rcpc, dotprod, and crypto [3]. It has a 64 KiB L1 data cache and a 1024 KiB L2 unified cache per core, and a 32 MiB shared L3 cache [3].

With this information, we are able to compute the theoretical machine peak of our setup. The Neon technology of the Neoverse-N1 core yields support for fused multiply-add on 2 128-bit vectors, corresponding to 2 sets of 2 double-precision floating-point fields. The theoretical machine peak becomes $2 \times 2 \times 2$ [mul, add] $\times\, 2.5$ [GHz] $= 20$ GFLOPS per core.

2.3.2 Software

For software, we retrieved the latest versions of LLVM, BLIS and ArmPL available at the time.

LLVM

The procedures for building MLIR and Clang were based on MLIR's *Getting Started* [36], Clang's *Getting Started: Building and Running Clang* [9] and the *Requirements* section of LLVM's *Getting Started with the LLVM System* [30], using the commit presented in listing 2.8 below [link].

Listing 2.8: `git log` output from the LLVM repository.

```
~/repos/llvm-project$ git log
commit 5f230ed762de050317a12bba56aadf8826a9b085 (HEAD -> main, origin/main, origin/HEAD)
Author: Tobias Gysi <tobias.gysi@nextsilicon.com>
Date:   Wed Aug 30 12:52:46 2023 +0000
...
```

BLIS

The procedure for building BLIS was based on the instructions in their GitHub repository [20], using the commit presented in listing 2.9 below [link].

[3]An EC2 instance provides certain memory bandwidth and number of virtual CPUs (vCPUs) depending on instance type. A c6g.metal instance has 64 vCPUs and 128 GiB of memory [53]. However, the end-user may customize the number of CPU cores and threads per core [55]. For a Graviton-based instance, each vCPU is a core of the CPU [54].

Listing 2.9: `git log` output from the BLIS repository.

```
~/repos/blis$ git log
commit 6dcf7666eff14348e82fbc2750be4b199321e1b9 (HEAD -> master, origin/master, origin/
    HEAD)
Author: Field G. Van Zee <field@cs.utexas.edu>
Date:   Sun Aug 27 14:18:57 2023 -0500
...
```

ArmPL

Free Arm Performance Libraries (version 23.04.1) was retrieved from *Product Download Hub* of the Arm Developer site [link], and installed according to provided Linux instructions [28].

Chapter 3

Initial DGEMM Benchmarks

Going forward, we will be implementing and running a double-precision matrix multiplication operation in various ways. The operation corresponds to equation 1.1 with A, B and C dimensioned $M \times K$, $K \times N$ and $M \times N$, respectively. For this master's thesis, we have chosen the dimensions $M = 2088$ and $N = K = 2048$ as they produce relatively large matrices suitable for tiling. Notably, 2088 is divisible by 2, 3, 4, 6, 8, 9, 12 etc.

The aim for this chapter is to produce an approximate lower and upper bound in DGEMM performance by assessing general-purpose compilers and high-performance compute libraries. As for benchmark programs, we have retrieved the `examples/matmul/matmul.c`[1] and `examples/matmul/matmul.blas.c`[2] files from Uday Bondhugula's Pluto repository [link] and made some modifications in line with our analysis. This includes:

- Adjusting the `M` macro to 2088.
- Modifying the `init_array()` and `init_matrices()` functions (since `M != N`).
- Removing any use of the MKL and OpenBLAS libraries.
- Introducing a corresponding benchmark for BLIS.

We present a single run of each benchmark just to give a reasonable idea of what current compilers and libraries are able to achieve.

3.1 General-Purpose Compilers

We will be compiling a naive-nest implementation in C using Clang and GCC. The naive-nest implementation refers to a loop nest iterating over M, then N and then K, as presented in listing 3.1.

[1] As of commit with hash cfb0139eb344fe9dd051c49079f95d1a2c20c988 [link].
[2] As of commit with hash d21758b727075581d63e383e36e1cdc72ecb5428 [link].

Listing 3.1: Naive-nest implementation of equation 1.1 with $M = 2088$ and $N = K = 2048$, in abbreviated pseudocode.

```
for i = 0 to 2088
    for j = 0 to 2048
        for k = 0 to 2048
            /* load %A[i][k],
                %B[k][j],
                %C[i][j] */
            /* mul, add */
            /* store %C[i][j] */
```

One can expect that Clang and GCC will perform quite poorly due to their general-purpose nature, but especially the former given previous results [5]. We compile with `-O3` which applies a number of optimizations that are intended to increase the execution speed [7, 18], and `-ffast-math` which essentially enables ignoring some of the IEEE, ISO or IEC rules/specifications regarding math functions, yielding improved performance for programs that do not require the guarantees of these specifications [8, 18].

Listing 3.2 demonstrates the compilation and execution of the naive-nest benchmark with Clang,

Listing 3.2: Executing the naive-nest benchmark with `clang -O3 -ffast-math`.

```
$ clang –v
clang version 18.0.0 (https://github.com/llvm/llvm–project.git 5
    f230ed762de050317a12bba56aadf8826a9b085)
Target: AArch64–unknown–linux–gnu

$ clang –O3 –ffast–math –DTIME input/matmul.c –o output/matmul.clang –lm
$ output/matmul.clang
45.371785s
0.39 GFLOPS
```

and listing 3.3 with GCC.

Listing 3.3: Executing the naive-nest benchmark with `gcc -O3 -ffast-math`.

```
$ gcc ––version
gcc (Ubuntu 11.4.0–1ubuntu1~22.04) 11.4.0

$ gcc –O3 –ffast–math –DTIME input/matmul.c –o output/matmul.gcc –lm
$ output/matmul.gcc
3.952853s
4.43 GFLOPS
```

Clang and GCC perform at **2.0%** and **22.2%** of the theoretical machine peak, respectively. Similar to the results of Uday Bondhugula [5], there is around a 10x performance gap between Clang and GCC. As observed, the execution time of a program is highly dependent of compiler, rather than programming language. This performance gap indicates that the LLVM middle and back ends are inferior to GCC's for optimizing GEMM applications.

It should be noted that a simple loop interchange from ijk to ikj drastically improves the performance of both compilers, but especially Clang as seen in listing 3.4,

Listing 3.4: Executing the modified naive-nest benchmark with `clang -O3 -ffast-math`.

```
$ clang –O3 –ffast–math –DTIME input/matmul_ikj.c –o output/matmul_ikj.clang –lm
$ output/matmul_ikj.clang
5.296322s
3.31 GFLOPS
```

and listing 3.5 for GCC.

Listing 3.5: Executing the modified naive-nest benchmark with `gcc -O3 -ffast-math`.

```
$ gcc -O3 -ffast-math -DTIME input/matmul_ikj.c -o output/matmul_ikj.gcc -lm
$ output/matmul_ikj.gcc
3.251577s
5.39 GFLOPS
```

This loop interchange boosts the performances to **16.6%** and **26.7%**, respectively. The near 9x improvement in Clang from a simple transformation gives insight to the inabilities of Clang and/or LLVM in terms of loop nest optimizations.

3.2 High-Perfomance Compute Libraries

Moving on to BLIS and ArmPL, we are going to demonstrate some of the highest execution speeds currently achievable by the state-of-the-art. We choose to compile with GCC, given that it has proven to be more suitable for GEMM applications in the previous section.

For BLIS, we are using the `bli_dgemm` function [link], for which the benchmark is compiled and executed in listing 3.6.

Listing 3.6: Executing the BLIS benchmark with `gcc -O3`.

```
$ gcc -DBLIS -O3 -DTIME input/matmul.blas.c -lblis -o output/matmul.blis
$ output/matmul.blis
0.944253s
18.55 GFLOPS
```

Then for ArmPL, we are using the `cblas_dgemm` function [link], for which the benchmark is compiled and executed in listing 3.7.

Listing 3.7: Executing the ArmPL benchmark with `gcc -O3`.

```
$ gcc -DARMPL -O3 -DTIME input/matmul.blas.c -larmpl -o output/matmul.armpl
$ output/matmul.armpl
0.943996s
18.55 GFLOPS
```

Both BLIS and ArmPL execute in less than a second, reaching **93.0%** of the theoretical machine peak.

Chapter 4

Exploring the MLIR Framework

MLIR is a framework for compiler development and is not intended to be used as a programming language as-is. However, for the purposes of this thesis, we will be implementing, compiling and executing a double-precision matrix multiplication operation with syntax and resources native to the MLIR project to demonstrate basic functionality. The tools to use for this purpose are `mlir-opt`[1] and `mlir-cpu-runner` which in conjunction lower the MLIR to LLVM IR and run it through the LLVM pipeline. As in chapter 3, we base the input MLIR off of equation 1.1 with A, B and C dimensioned $M \times K$, $K \times N$ and $M \times N$, respectively, and choose $M = 2088$ and $N = K = 2048$ for continuity. The benchmark program can be seen in listing A.1 in the appendix, where the `@matmul` function corresponds to the naive-nest implementation as presented in listing 3.1. Any mention of the `@matmul` function in this thesis refers to the function in the benchmark program.

We lower the benchmark program to the `llvm` dialect using a sequence of various passes and pipe the output to `mlir-cpu-runner -O3`, as presented in listing 4.1 below.

Listing 4.1: Executing the `@matmul` benchmark with no optimizations beside `-O3`.

```
$ mlir-opt --convert-linalg-to-loops --lower-affine --convert-scf-to-cf --convert-cf-to-llvm
    --convert-arith-to-llvm --convert-func-to-llvm --expand-strided-metadata --finalize-
    memref-to-llvm --reconcile-unrealized-casts input/matmul.mlir | mlir-cpu-runner -O3 -
    e main --entry-point-result=void --shared-libs=lib/libmlir_runner_utils.so
0.346990 GFLOPS
```

The performance is at a poor 1.7% of the theoretical machine peak. This was expected, given the previous demonstration of Clang; an unoptimized naive-nest implementation does not perform well.

[1]The main purpose of `mlir-opt` is to test compiler passes.

4.1 MLIR's Infrastructure for Optimizations

In listing 4.1 above, one can note that `-convert-linalg-to-loops`, `-lower-affine`, `-convert-scf-to-cf`, `-convert-cf-to-llvm`, `-convert-arith-to-llvm`, `-convert-func-to-llvm`, `-expand-strided-metadata`, `-finalize-memref-to-llvm` and `-reconcile-unrealized-casts` were input to `mlir-opt` alongside the input file. The purpose of these passes are only to lower the source code to the `llvm` dialect, meaning there were no direct optimization directives. The passes needed for this purpose, particularly the `-convert-*-to-llvm` passes, directly depend on what dialects are present in the IR throughout the progressive lowering. Additionally, the `-expand-strided-metadata` pass needs to be applied before the `-finalize-memref-to-llvm`[2] pass [11] and `-reconcile-unrealized-casts` needs to be applied as a last step [63]. Given that `@matmul` is a nest of `affine` loops (see listing A.1 in the appendix), the optimizations most relevant to our code likely relates to loop transformations or the `affine` dialect. We will focus on the `-affine-loop-tile` and `-affine-data-copy-generate` passes, which we will use to implement loop tiling and explicit copying.

4.1.1 Loop Tiling

As there exist many tiling schemes, the first thing to investigate is the functionality of `-affine-loop-tile`. By applying this pass to `@matmul` with various configurations, we find that the tiling scheme of `-affine-loop-tile` follows the general structure of listing 4.2.

Listing 4.2: `@matmul` after applying the `-affine-loop-tile` pass, in abbreviated pseudocode.

```
for i = 0 to 2088 step M_B
    for j = 0 to 2048 step N_B
        for k = 0 to 2048 step K_B
            for ii = i to i + M_B
                for jj = j to j + N_B
                     for kk = k to k + K_B
                        /* load %A[ii][kk], %B[kk][jj], %C[ii][jj] */
                        /* mul, add */
                        /* store C[ii][jj]*/
```

The tiling scheme implemented in MLIR is quite straight-forward; one additional outer loop is added for each of the original loops, in order of innermost to outermost. The tile sizes, which we will refer to as M_B, N_B and K_B, can be directly manipulated through the `-tile-size` and `-tile-sizes` options, but we disregard the `-tile-size` option since it adds the constraint $M_B = N_B = K_B$. In choosing tile sizes, we must consider the cache level capacities of our setup, as well as the structure of the generated loop nest. Given the tile sizes M_B, N_B and K_B, we are partitioning matrices A, B and C into tiles dimensioned $M_B \times K_B$, $K_B \times N_B$ and $M_B \times N_B$, respectively. As the capacity of cache is limited, this leads into the

[2]The `-finalize-memref-to-llvm` pass was previously named `-convert-memref-to-llvm`, but was renamed as the `-expand-strided-metadata` pass must be run beforehand [11, 10].

natural constraint that the tile sizes cannot be too large. More specifically,

$$\begin{cases} M_B \cdot K_B \cdot \frac{8}{1024} \text{ KiB} & <= 64 \text{ KiB or } 1024 \text{ KiB} \\ K_B \cdot N_B \cdot \frac{8}{1024} \text{ KiB} & <= 64 \text{ KiB or } 1024 \text{ KiB} \\ M_B \cdot N_B \cdot \frac{8}{1024} \text{ KiB} & <= 64 \text{ KiB or } 1024 \text{ KiB} \\ (M_B \cdot K_B + K_B \cdot N_B + M_B \cdot N_B) \cdot \frac{8}{1024} \text{ KiB} & <= 1024 \text{ KiB} \end{cases} \tag{4.1}$$

where the RHS of each line assume we are only considering tiling for the L1 and L2 cache levels, bound to 64 and 1024 KiB, respectively.

The first dimension to be traversed in both the inner and outer loop nests is the k-dimension, meaning the C-tile is the longest to persist in cache since each access to C is invariant of k. While processing a single C-tile, K/K_B A-tiles, or $M_B \cdot K$ entries of A, and K/K_B B-tiles, or $K \cdot N_B$ entries of B, are streamed through cache. The second dimension to be traversed is the j-dimension. This means we will traverse through the entire B matrix, but start over on the same K/K_B tiles of A N/N_B times over. In the next `ii`-iteration, we move over to the next set of K/K_B tiles of A and traverse B once more. In total, we find that A is read N/N_B times, B is read M/M_B times and C is read and stored K/K_B times. The total number of floating-point memory transfers is thus determined by equation 4.2.

$$(M \cdot N) \cdot 2 \cdot K/K_B + (K \cdot N) \cdot M/M_B + (M \cdot K) \cdot N/N_B \tag{4.2}$$

Moreover, an A-tile is reused N_B times, a B-tile M_B times, and a C-tile K_B times. So, the first takeaway is that M_B, N_B and K_B should all be maximized to decrease the number of memory transfers and increase reuse. Another important aspect to take into consideration is the size of respective tile in relation to cache capacities. This way, we can generally manipulate how and where each tile is placed in or streamed through the cache hierarchy. Constraining our analysis to only consider the L1 and L2 cache levels and tile sizes that are even divisors of either M $(= 2088)$, N $(= 2048)$ or K $(= 2048)$, we find that $M_B = 232$, $N_B = 512$ and $K_B = 8$ to be a suitable choice. Inserting these into equation 4.1 yields equation 4.3.

$$\begin{cases} 232 \cdot 8 \cdot \frac{8}{1024} \text{ KiB} & = 14.5 \text{ KiB} \\ 8 \cdot 512 \cdot \frac{8}{1024} \text{ KiB} & = 32.0 \text{ KiB} \\ 232 \cdot 512 \cdot \frac{8}{1024} \text{ KiB} & = 928.0 \text{ KiB} \\ (232 \cdot 8 + 8 \cdot 512 + 232 \cdot 512) \cdot \frac{8}{1024} \text{ KiB} & = 974.5 \text{ KiB} \end{cases} \tag{4.3}$$

The A- and B-tiles occupy around 22.7% and 50.0% of L1 cache, respectively, and the C-tile occupy around 90.6% of L2 cache. We have essentially maxed out the cache capacity, with space for one C-tile at a time and limited, but sufficient, space for tiles of A and B to flow through the cache hierarchy. This is illustrated in figure 4.1.

As seen in listing 4.3, this loop tiling configuration yields a 12x improvement. At 20.5% of the theoretical machine peak, we are now on par with the performance of GCC.

Listing 4.3: Executing the `@matmul` benchmark with loop tiling applied.

```
$ mlir—opt —affine—loop—tile="tile—sizes=232,512,8" —convert—linalg—to—loops —lower—
    affine —convert—scf—to—cf —convert—cf—to—llvm —convert—arith—to—llvm —convert—func—
    to—llvm —expand—strided—metadata —finalize—memref—to—llvm —reconcile—unrealized—
    casts input/matmul.mlir | mlir—cpu—runner —O3 —e main —entry—point—result=void —
    shared—libs=lib/libmlir_runner_utils.so
4.100237 GFLOPS
```

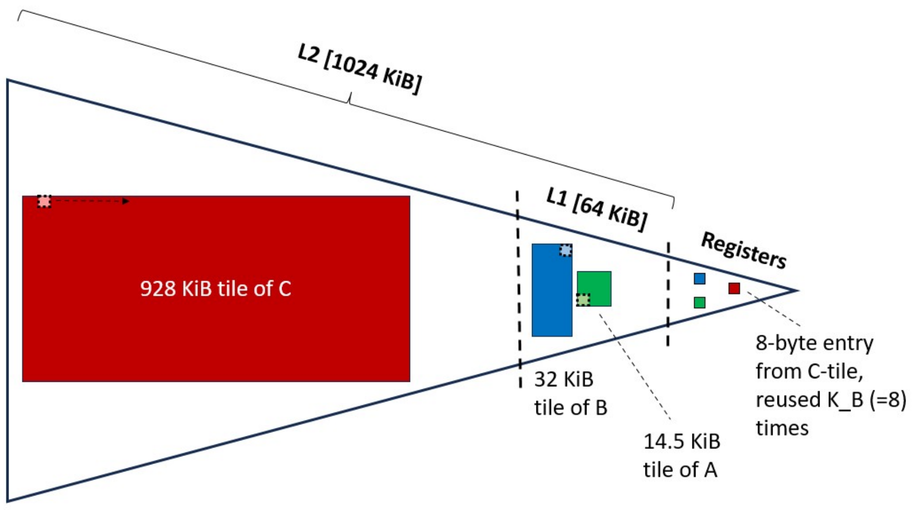


Figure 4.1: Illustration of the A- (green), B- (blue) and C-tiles (red) in relation to L1 and L2 cache and registers.

4.1.2 Explicit Copying

Considering the size of the tiles (see equation 4.3), we know that one tile of each matrix is accessed in memory at a time. The data within these tiles are not contiguous in memory, leading to conflict misses, TLB misses, and more prefetch streams [5]. Applying explicit copying, packing the A-, B- and C-tiles into contiguous buffers before they are being accessed and reused, will eliminate this issue and improve performance even further. Listing 4.4 demonstrates how the copying takes place.

Listing 4.4: `@matmul` after applying the `-affine-loop-tile` and `-affine-data-copy-generate` passes, in abbreviated pseudocode.

```
for i = 0 to 2088 step M_B
    for j = 0 to 2048 step N_B
        /* copy %C[i:i+M_B][j:j+N_B] to %C_C */
        for k = 0 to 2048 step K_B
            /* copy %A[i:i+M_B][k:k+K_B] to %A_C */
            /* copy %B[k:k+K_B][j:j+N_B] to %B_C */
            for ii = 0 to M_B
                for jj = 0 to N_B
                     for kk = 0 to K_B
                        /* load %A_C[ii][kk], %B_C[kk][jj], %C_C[ii][jj] */
                        /* mul, add */
                        /* store %C_C[ii][jj]*/
```

Note that the three inner loops as well as the copy statements as presented in listing 4.4 do not take into account the case where M_B, N_B or K_B are not divisors of respective matrix dimension. In MLIR, one can use `affine.map` and `affine.min` to handle such cases. E.g., for the case where $M_B = 300$, we would replace

```
ii = 0 to 300
```

in the loop header with

```
ii = 0 to affine.min affine_map<(d0) -> (2088 - d0, 300)> (i)
```

which assigns the upper bound with either `300` or `2088 - i`, depending on which is smaller. This ensures that `ii` does not go out of bounds when traversing an edge tile. Given our choice of parameters, the A-, B- and C-tiles are dimensioned 232×8, 8×512 and 232×512, respectively.

We are going to implement this scheme using the `-affine-data-copy-generate` pass. We guide the pass accordingly through the `-fast-mem-capacity` option. This option essentially limits the amount of memory that can be occupied by these buffers at once. Given our set constraints and the size of L2 cache, we know that a capacity of 1024 KiB will suffice regardless of choice of tile sizes. Listing C.3 in the appendix presents the resulting IR. As more transformations are applied, the output becomes long and hard to read, but on lines 15, 25 and 34 we note that memory is being allocated according to the specified tile sizes and at the right nest levels. Following the call to `memref.alloc()` is a loop nest to copy each element into the newly allocated `memref` iteratively. These new matrix tiles replace the uses of the original operands within the inner loop body.

Applying explicit copying in conjunction with loop tiling boosts performance to 34.6%, as seen in listing 4.5.

Listing 4.5: Executing the `@matmul` benchmark with loop tiling and explicit copying applied.

```
$ mlir–opt –affine–loop–tile="tile–sizes=232,512,8" –affine–data–copy–generate="generate
    –dma=false fast–mem–space=0 fast–mem–capacity=1024" –convert–linalg–to–loops –lower–
    affine –convert–scf–to–cf –convert–cf–to–llvm –convert–arith–to–llvm –convert–func–
    to–llvm –expand–strided–metadata –finalize–memref–to–llvm –reconcile–unrealized–
    casts input/matmul.mlir | mlir–cpu–runner –O3 –e main –entry–point–result=void –
    shared–libs=lib/libmlir_runner_utils.so
6.928518 GFLOPS
```

We have finally surpassed the performance of GCC, even when manually optimized with a loop interchange. However, we must improve the performance by almost 3x to be on par with BLIS and ArmPL.

Moreover, one could use the `-affine-loop-unroll`, `-affine-scalrep` and `-affine-super-vectorize` passes to apply loop unrolling, scalar replacement and vectorization, respectively. Our attempts at applying these to `@matmul` on top of `-affine-loop-tile` and `-affine-data-copy-generate` did however not improve performance.

Chapter 5

Implementing a High-Performance DGEMM Routine in MLIR

Constraining ourselves to the `-affine-loop-tile` and `-affine-data-copy-generate` passes produced a performance far from that of BLIS and ArmPL. Given similar matrix multiplication experiments that produced near-peak performance [5, 62], we believe that there is room for improvement. Since the MLIR project presents itself as an extensible framework with fully extensible operations [37, 38], the natural next step is to build our own extension of MLIR with the ultimate purpose of improving the performance of `@matmul`. This chapter will elaborate on the notion of extensibility by demonstrating the implementation of an MLIR dialect and operation. We draw inspiration from StableHLO of the OpenXLA project, based on the MHLO dialect [link], which functions as a portability layer between machine learning frameworks and compilers [48, 50].

The goal for this section is to implement the `@matmul` function as a functioning `mlir::Op` in MLIR. Our approach is to implement a dialect `hlo` (derived from *High-Level linear algebra Operations*) with a single `mlir::Op` called `matmul`, with the same functionality of the previously defined `@matmul` function. We will achieve this by implementing a pass `-hlo-matmul-to-loops` which expands the operation according to the specification of `@matmul` through *pattern rewriting* [link]. MLIR provides some general guidelines for how to go about implementing a dialect, such as the *Defining Dialects* documentation [link] and the *Creating a Dialect* tutorial [link]. Relevant source files and the general directory structure, including file names, can be found in appendix B.1.2.

5.1 HLO: A Minimal Dialect for High-Level Operations

We are able to minimize the amount of manually-typed code through the usage of LLVM's TableGen language. In each `.td` file, we declaratively define fundamental aspects of `hlo` (see listings B.2 and B.3 in the appendix). Then, in `mlir/include/.../IR/CMakeLists.txt` we add

```
add_mlir_dialect(HLOOps hlo)
```

which generates files for e.g. `Op` and dialect declarations and definitions, based on `HLOBase.td` and `HLOOps.td`. These are then imported to corresponding implementation files. Using this approach, `HLODialect.cpp` and `HLOOps.cpp` only require a few lines of code to implement `mlir::hlo::HLODialect::initialize()` and `mlir::hlo::HLODialect::registerOperations()`.

Finally, we make small adjustments to `mlir/include/mlir/initAllDialects.h`, so that `hlo` is recognized by `mlir-opt`. This essentially means that `mlir-opt` will not raise an error if we input a file containing the line

```
hlo.matmul ins(%A : memref<2088x2048xf64>, %B : memref<2048x2048xf64>, %C : memref
    <20882048xf64>)
```

but at this point there exists no functionality to execute it. This will be handled by `-hlo-matmul-to-loops`.

5.2 Building the Loop Nest

The `-hlo-matmul-to-loops` pass is declared in `Passes.td` (see listing B.4 in the appendix), where we specify the `mlir::hlo::createHLOMatmulToLoopsPass()` function to be used as a constructor. It is also in here we declare the available options:

- `tile` – enable loop tiling.
- `tile-params` - loop tile parameters M_C, K_C, M_R and N_R.
- `copy` – enable explicit copying.
- `unroll` – enable loop unrolling.
- `unroll-factor` – unroll factor for the `kk`-loop.
- `scalar-replace` – enable scalar replacement.
- `vectorize` – enable vectorization.

for which respective value is specified through e.g. `-hlo-matmul-to-loops="tile=true"`.

The minimum requirement for implementing our pass is thus to define its constructor `mlir::hlo::createHLOMatmulToLoopsPass()`, with return type `std::unique_ptr<OperationPass<func::FuncOp»`. This constructor simply consists of a return statement

with `std::make_unique<HLOMatmulToLoops>()`, meaning the majority of the implementation resides within the `HLOMatmulToLoops` class, where we must define the inherited virtual function `runOnOperation()`. This is the function that acts upon input MLIR operations. In our case, any occurrence of `hlo.matmul` is transformed by a rewrite pattern `HLO-MatmulRewritePattern`, derived from `mlir::OpRewritePattern<hlo::MatmulOp>`, with a single function `HLOMatmulRewritePattern::matchAndRewrite` with arguments `hlo::MatmulOp op` and `PatternRewriter &rewriter`, returning an instance of `Logi-calResult` signaling either a success or failure. An abbreviated version of `HLOMatmulToLoops.cpp` is presented in listing B.5 in the appendix, giving an overview of how each class and function relate to each other. One can note that `HLOMatmulToLoops` directly inherits from `mlir::hlo::HLOMatmulToLoopsBase<HLOMatmulToLoops>`, which has not been previously mentioned. This is a TableGen'erated class defined in `include/mlir/Dialect/HLO/Transforms/Passes.h.inc`, inheriting from the `mlir::OperationPass<func::FuncOp>` class. This class provides some basic functionality, such as processing user input, but the majority of `HLOMatmulToLoops` is manually implemented. At the most basic level, in the case where every optimization is disabled, the implementation is minimal; we use the `rewriter` and its `create` function to construct a loop nest structure as presented in listing 3.1. We set the correct upper bounds for each `for`-loop by reading the shapes of each operand passed to `op`.

The final step is to make small adjustments to `mlir/include/mlir/initAllPasses.h`, so that `-hlo-matmul-to-loops` is recognized by `mlir-opt`. Listings C.4 and C.5 in the appendix demonstrates that `-hlo-matmul-to-loops`, with no optimizations enabled, functions properly. Note that `hlo.matmul`, lowered with no optimizations enabled, differs from `@matmul` in that the `fast` option [link] is enabled in the calls to `arith.mulf` and `arith.addf` (compare listing C.6 and `@matmul` of listing A.1 in the appendix).

We create a new benchmark for `hlo.matmul` (see listing A.2 in the appendix) and run it with `mlir-opt` and `mlir-cpu-runner -O3`, as seen in listing 5.1. Again, this naive-nest implementation is on par with the performance of Clang.

Listing 5.1: Executing the `hlo.matmul` benchmark with no optimizations beside `-O3`.

```
$ mlir—opt —hlo—matmul—to—loops —convert—linalg—to—loops —lower—affine —convert—scf—to—
    cf —convert—cf—to—llvm —convert—arith—to—llvm —convert—func—to—llvm —expand—strided—
    metadata —finalize—memref—to—llvm="use—aligned—alloc=true" —reconcile—unrealized—
    casts —canonicalize input/hlo_matmul.mlir | mlir—cpu—runner —O3 —e main —entry—point
    —result=void —shared—libs=lib/libmlir_runner_utils.so —shared—libs=lib/
    libmlir_c_runner_utils.so
0.389302 GFLOPS
```

5.3 Optimizing the Loop Nest

At this point we have managed to effectively recreate the `@matmul` function using a compiler approach. Going forward, we are going to explore the implementation and effect of common optimization techniques.

Using loop tiling, explicit copying and loop unrolling we aim to encompass a variation of the matrix multiplication algorithm described by the BLIS developers [60], based on the GEMM design principles laid out by Kazushige Goto and Robert A. van de Gejin from the University of Texas at Austin in 2008 [19]. Using our own naming convention, the ***A*** and ***B*** matrices are first partitioned along K with a cache block size K_C, producing $M \times K_C$ and $K_C \times N$ tiles of A and B, respectively. The B-tile, B_C, is packed to a contiguous buffer. The C matrix as well as the A-tile is further partitioned along M with a cache block size M_C and the resulting $M_C \times K_C$ tile of A, A_C, is packed to a contiguous buffer. The last partitions are made along N, M_C and K_C with sizes N_R, M_R and 1, respectively. We refer to the $M_R \times K_C$-, $K_C \times N_R$- and $M_R \times N_R$-sized tiles produced by the partitioning along N and M_C as A_R, B_R and C_R, respectively. The innermost computation iterating over K_C is referred to as a *micro-kernel* [60], where C_R, a *register tile* (composed of CPU registers) of C, is updated. During the operation, memory is transferred so that C resides in main memory, B_C resides in L3 cache, A_C in L2 cache and B_R in L1 cache [33]. Our implementation will follow this schedule with the exception that B_R is packed to a contiguous buffer instead of B_C, meaning we choose not to control memory allocation beyond the L2 cache level. This choice was made based on the approach employed in [5]. It should be noted that BLIS implements GEMM with an additional partition along N which reduces the size of B_C to $K_C \times N_C$ [33]. This additional partition is not necessary in our implementation given that we only consider the L1 and L2 cache levels. Given the placement of each tile in memory, the choice of K_C, M_C and N_R pertain to the capacity of cache and N_R and M_R to the capacity of CPU registers.

The resulting loop nest is illustrated in figure 5.1. The labels, K_C, M_C, N_R and M_R, indicate which partition is made at what level and the A_C and B_R labels indicate where the packing takes place. The outermost box illustrates the partitioning of the A and B matrices along K with K_C. The innermost box corresponds to the micro-kernel, which iterates over K_C and through instances of A_R and B_R. The tiles are color-coordinated according to their prescribed placement in memory — red for CPU registers, blue for L1 cache, green for L2 cache, and white for L3 cache or main memory. Furthermore, the subcomputation within the micro-kernel — one round of updating the elements of the register tile of C — operates on $M_R \times 1$ and $1 \times N_R$ panels of A_R and B_R, respectively, which are loaded to CPU registers [33].

The way we will go about implementing the optimizations is based on the demonstrated approach in [5]. We are going to recreate the loop nest structures presented [5] as a precaution to ensure that the implementation steps ultimately result in the correct design. Loop tiling will be used to partition the unoptimized version of `hlo.matmul` accordingly, explicit copying to pack A_C and B_R to contiguous buffers, and loop unrolling to remove the redundant loops that are generated from loop tiling alone. These loops, which are created from partitioning along N and M_C, are redundant in the sense that the resulting loop nest updates one element of the register tile in each iteration instead of the entire register tile, which is necessary to constitute the micro-kernel. Loop unrolling will also be used to unroll the micro-kernel by a factor K_U, in an attempt to improve performance further. In addition to loop tiling, explicit copying and loop unrolling, we will implement scalar replacement to remove redundant loads that occur from loop unrolling and vectorization to target SIMD instructions. In terms of code, implementing these optimizations means for the most part to extend the `HLOMatmulToLoops` class, especially the `HLOMatmulRewritePattern ::matchAndRewrite` function.

5.3.1 Loop Tiling

Listing 5.2 below presents the loop tiling scheme we are going to implement.

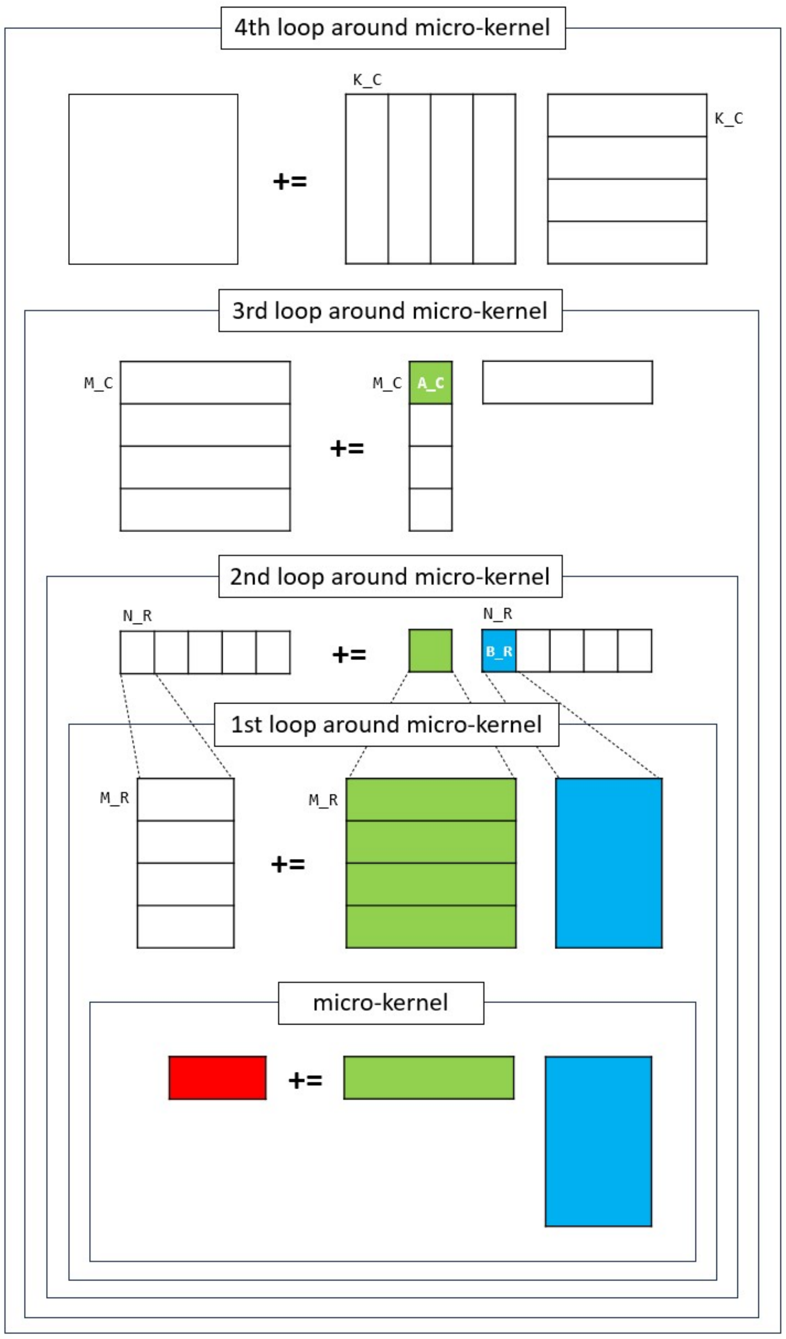

4th loop around micro-kernel
K_C
+=
K_C
3rd loop around micro-kernel
M_C
M_C
A_C
+=
2nd loop around micro-kernel
N_R
N_R
+=
B_R
1st loop around micro-kernel
M_R
M_R
+=
micro-kernel
+=

Figure 5.1: A schematic representation of `hlo.matmul` optimized according to our adaptation of the matrix multiplication algorithm described by the BLIS developers [60].

Listing 5.2: `hlo.matmul` after enabling loop tiling, in abbreviated pseudocode.

```
for k = 0 to 2048 ceildiv K_C
    for i = 0 to 2088 ceildiv M_C
        for jj = 0 to 2048 ceildiv N_R
            for ii = i * M_C/M_R to min (M/M_R, i * M_C/M_R + M_C/M_R)
                for kk = 0 to min (K_C, k * -K_C + K)
                     for jjR = 0 to N_R
                        for iiR = 0 to M_R
                            /* load %A[ii * M_R + iiR][k * K_C + kk],
                                    %B[k * K_C + kk][jj * N_R + jjR],
                                    %C[ii * M_R + iiR][jj * N_R + jjR] */
                            /* mul, add */
                            /* store %C[ii * M_R + iiR][jj * N_R + jjR] */
```

Compare this loop tiling scheme to the illustration in figure 5.1; the `k`-, `i`-, `jj`-, `ii`- and `kk`-loops correspond to the boxes from outermost to innermost. Variable `k` iterates through the inner dimensions of A and B, `i` through the A_C tiles along the specified `K_C`-wide panel of A, `ii` and `jj` through the A_R tiles of A_C, the B_R tiles along the specified `K_C`-wide panel of B and the register tiles along the specified $M_C \times N_R$-sized tile of C, `kk` through the inner dimensions of A_R and B_R, and `iiR` and `jjR` through the outer dimensions of A_R and B_R as well as through the rows and columns of C_R. The upper bounds of the `ii`- and `kk`-loops are not constant, meaning that M_C and K_C are not bound to being divisors of M and K, respectively. The upper bounds of the `jjR`- and `iiR`-loop must however be constant. This constraint arises from the loop unrolling that is going to be applied in a later stage. As such, N_R must be a divisor of N and M_R a divisor of both M_C and M. These constraints are summarized in equation 5.1 below.

$$\begin{cases} N_R \mid N \\ M_R \mid M \\ M_R \mid M_C \end{cases} \tag{5.1}$$

Given the existence of the `-affine-loop-tile` pass, the MLIR framework already possesses resources to carry out loop tiling. For our purposes, we are going to use `mlir::affine::tile` as well as `mlir::affine::interchangeLoops` in `HLOMatmulToLoops` to achieve the general loop nest structure of listing 5.2. The starting point is a nest of the `i`-, `j`- and `k`-loops, as presented in listing 3.1. We begin by placing the `k`-loop as the outermost loop by interchanging the `j`- and `k`-loops, following up with another loop-interchange of the `i`- and `k`-loops. Then, we apply `mlir::affine::tile` to an array of the `i`-, `k`- and `j`-loops, in that order, specifying M_C, K_C and N_R as tile sizes. After this round of tiling, the `j`-loop has been transformed into the `jj`-loop and the `ii`-, `kk` and `jjR`-loops have been added to the nest. Since the partition along M_C has not yet been done, the current loop nest is missing the `iiR`-loop. We create this loop by applying `mlir::affine::tile` once more, but only to the output `ii`-loop, specifying a tile size of M_R. This produces the output presented in listing C.7 in the appendix.

This intermediate result practically follows our set design, but we must make a couple of adjustments to the loop bounds in order to recreate the loop nest structure presented in the *Tiling in MLIR* section of [5]. Firstly, for each of the three innermost loops, we want to shift the bounds so that the lower bound is a constant zero. We also have to modify every use of the loop's induction variable, given that its value range is changed. For the following made up example,

```
#lowerBound = affine.map<(d0) -> (d0)>
#upperBound = affine.map<(d0) -> (120, d0 + 18)>
...
affine.for %arg0 = #lowerBound(%idx) to min #upperBound(%idx) {
    %0 = arith.addi %arg0, %arg0 : i64
}
```

this modification would produce

```
#upperBound = affine.map<(d0) -> (120 - d0, 18)>
...
affine.for %arg0 = 0 to min #upperBound(%idx) {
    %0 = affine.apply affine.map<(d0, d1) -> (d0 + d1)>(%arg0, %idx)
    %1 = arith.addi %0, %0 : i64
}
```

`%arg0` has been replaced by `%0` using `mlir::Value::replaceUsesWithIf`, excluding the use of `%arg0` within the `affine.apply` call. We apply this modification to the three innermost `for`-loops, in order of innermost to outermost.

Secondly, we can note that `mlir::affine::tile` tiles a `for`-loop by simply setting the step size to the specified tile size. This does not align with our design, where we instead iterate through tile indices, e.g., the $A[0 : M_C, 0 : K_C]$ section corresponds to A_C with index $(k = 0, i = 0)$ and the $A[M_C : 2 \cdot M_C, 0 : K_C]$ section corresponds to A_C with index $(k = 0, i = 1)$. We can easily remove a `step` by dividing the lower and upper bounds by its original step size, and then calling `mlir::affine::AffineForOp::setStep(1)`. Again, we also have to modify every use of the loop's induction variable. For the following made up example,

```
#lowerBound = affine.map<(d0) -> (d0)>
#upperBound = affine.map<(d0) -> (120, d0 + 18)>
...
affine.for %arg0 = #lowerBound(%idx) to min #upperBound(%idx) step 3 {
    %0 = arith.addi %arg0, %arg0 : i64
}
```

this modification would produce

```
#lowerBound = affine.map<(d0) -> (d0 ceildiv 3)>
#upperBound = affine.map<((d0) -> (40, d0 ceildiv 3 + 6)>
...
affine.for %arg0 = max #lowerBound(%idx) to min #upperBound(%idx) {
    %0 = affine.apply affine.map<(d0) -> (d0 * 3)>(%arg0)
    %1 = arith.addi %0, %0 : i64
}
```

We apply this modification to the four outermost `for`-loops, in order of innermost to outermost. This results in MLIR with the correct tiling strategy, as presented in listing C.8 in the appendix.

Loop tiling alone generates the `jjR`- and `iiR`-loops, which means the current loop nest updates one element of C_R in each iteration (see listing 5.2). Consequently, we do not have any control over register allocation at this stage. Thus, we are going to temporarily disregard the choice of M_R (arbitrarily set to 3) and choose K_C, M_C and N_R for cache tiling. Given that A_C resides in L2 and B_R in L1 cache [33], we are constrained by equation 5.2,

$$\begin{cases} M_C \cdot K_C \cdot \frac{8}{1024} \text{ KiB} & <= 1024 \text{ KiB} \\ K_C \cdot N_R \cdot \frac{8}{1024} \text{ KiB} & <= 64 \text{ KiB} \\ (M_C \cdot K_C + K_C \cdot N_R) \cdot \frac{8}{1024} \text{ KiB} & <= 1024 \text{ KiB} \end{cases} \tag{5.2}$$

as well as equation 5.1. Analyzing the loop nest (see listing 5.2 and figure 5.1), we discern that each instance of A_C is reused N/N_R times, that each instance of B_R is loaded M/M_C times and reused M_C/M_R times, and that each instance of C_R is reused K_C times. Thus, K_C and M_C should be assigned with large values and N_R (and M_R) with a small value to decrease the number of memory transfers and increase reuse. By trying different values and calculating the occupied space in cache, we find $M_C = 165$, $K_C = 720$, and $N_R = 8$ to be a suitable choice. Inserting these values into equation 5.2 yields equation 5.3,

$$\begin{cases} 165 \cdot 720 \cdot \frac{8}{1024} \text{ KiB} & = 928.125 \text{ KiB} \\ 720 \cdot 8 \cdot \frac{8}{1024} \text{ KiB} & = 45.0 \text{ KiB} \\ (165 \cdot 720 + 720 \cdot 8) \cdot \frac{8}{1024} \text{ KiB} & = 973.125 \text{ KiB} \end{cases} \tag{5.3}$$

meaning A_C occupies 928.125 KiB of L2 cache and B_R occupies 45.0 KiB of L1 cache, thus resembling our previous cache tiling setup (see equation 4.3). With this configuration, applying loop tiling leads to a performance at 9.7% of the theoretical machine peak, as seen in listing 5.3.

Listing 5.3: Executing the `hlo.matmul` benchmark after applying loop tiling with $M_C = 165$, $K_C = 720$, $M_R = 3$, and $N_R = 8$

```
$ mlir—opt —hlo—matmul—to—loops="tile=true tile—params=165,720,3,8" —convert—linalg—to—
    loops —lower—affine —convert—scf—to—cf —convert—cf—to—llvm —convert—arith—to—llvm —
    convert—func—to—llvm —expand—strided—metadata —finalize—memref—to—llvm="use—aligned—
    alloc=true" —reconcile—unrealized—casts —canonicalize input/hlo_matmul.mlir | mlir—
    cpu—runner —O3 —e main —entry—point—result=void —shared—libs=lib/
    libmlir_runner_utils.so —shared—libs=lib/libmlir_c_runner_utils.so
1.936209 GFLOPS
```

5.3.2 Explicit Copying

This step involves packing each instance of A_C and B_R to contiguous buffers before they are accessed and reused. Listing 5.4 demonstrates how the packing shall take place.

Listing 5.4: `hlo.matmul` after enabling loop tiling and explicit copying, in abbreviated pseudocode.

```
for k = 0 to 2048 ceildiv K_C
    for i = 0 to 2088 ceildiv M_C
        /* copy %A[i * M_C : i * M_C + M_C][k * K_C : k * K_C + K_C]
           to %A_C */
        for jj = 0 to 2048 ceildiv N_R
            /* copy %B[k * K_C : k * K_C + K_C][j * N_R : j * N_R + N_R]
               to %B_R */
            for ii = i * M_C/M_R to min (M/M_R, i * M_C/M_R + M_C/M_R)
                for kk = 0 to min (K_C, k * -K_C + K)
                     for jjR = 0 to N_R
                        for iiR = 0 to M_R
                            /* load %A_C[-i * M_C + ii * M_R + iiR][kk],
                                    %B_R[kk][jjR],
                                    %C[ii * M_R + iiR][jj * N_R + jjR] */
                            /* mul, add */
                            /* store %C[ii * M_R + iiR][jj * N_R + jjR] */
```

We achieve this by using the underlying utility of the `-affine-data-copy-generate` pass, the `mlir::affine::affineDataCopyGenerate` function. This function has the same functionality as the `-affine-data-copy-generate` pass, with the added option to target a certain memory buffer (`%A`, `%B`, or `%C`). We are able to generate the correct copy procedures by specifying the arguments `mlir::affine::affineForOp forOp` and `std::optional< mlir::Value> filterMemRef`, and limiting the `fastMemCapacityBytes` option of the `mlir::affine::AffineCopyOptions &copyOptions` argument accordingly. We make two calls to the function, once for each of `%A` and `%B`, setting `forOp` to the immediately surrounding `for`-loop, `filterMemRef` to either `%A` or `%B`, and `fastMemCapacityBytes` to either $1024 \cdot 1024$, corresponding to the capacity of L2 cache in bytes, or $64 \cdot 1024$, corresponding to the capacity of L1 cache in bytes, in `HLOMatmulToLoops`. This produces the MLIR presented in listing C.9 in the appendix. Applying explicit copying improves performance by 3.3x, now 32.1% of the theoretical machine peak, as seen in listing 5.5.

Listing 5.5: Executing the `hlo.matmul` benchmark after applying loop tiling with $M_C = 165$, $K_C = 720$, $M_R = 3$ and $N_R = 8$, and explicit copying.

```
$ mlir-opt --hlo-matmul-to-loops="tile=true tile-params=165,720,3,8 copy=true" --convert-
    linalg-to-loops --lower-affine --convert-scf-to-cf --convert-cf-to-llvm --convert-arith-
    to-llvm --convert-func-to-llvm --expand-strided-metadata --finalize-memref-to-llvm="use
    -aligned-alloc=true" --reconcile-unrealized-casts --canonicalize input/hlo_matmul.mlir
     | mlir-cpu-runner -O3 -e main --entry-point-result=void --shared-libs=lib/
    libmlir_runner_utils.so --shared-libs=lib/libmlir_c_runner_utils.so
6.411241 GFLOPS
```

Despite the relatively poor performance when applying loop tiling, we are currently on par with the performance achieved from using `-affine-loop-tile` and `-affine-data-copy-generate`.

5.3.3 Loop Unrolling

Enabling the micro-kernel

We have not yet implemented the micro-kernel, which is enabled by fully unrolling the `jjR`- and `iiR`-loops. This effectivey means to replace the two innermost loops with the corresponding sequence of statements. We easily achieve this by making two calls to `mlir::affine::loopUnrollFull(mlir::affine forOp)`, one for each of the redundant loops, in `HLOMatmulToLoops`. The resulting loop nest corresponds to listing 5.4, but without the `jjR`- and `iiR`-loops, and the `kk`-loop as the innermost loop.

The N_R $(= 8)$ and M_R $(= 3)$ parameters control the number of registers used in each subcomputation within the micro-kernel. This is because $M_R \times 1$- and $1 \times N_R$-sized panels of A_R and B_R, respectively, are used to update an $M_R \times N_R$ register tile of C in one iteration of the `kk`-loop. Figure 5.2, which can be seen as an elaboration on the micro-kernel illustrated in figure 5.1, illustrates a hypothetical register allocation procedure within the micro-kernel for the elements loaded from A_C, B_R and C. Since the $M_R \cdot N_R$ values stored in the register tile are reused in the next `kk`-loop iteration, it follows that $M_R \cdot N_R$ registers, or $M_R \cdot N_R/2$ 2-length vector registers[1], are reserved for the register tile. For A_C and B_R, however, we instead hypothesize that M_R registers are reserved for M_R elements of A_C (one for each), and 1 register is reserved for N_R elements from B_R (one for all). This is because the `iiR`-loop is first to be unrolled, meaning the same M_R values from A_C are used over and over. The M_R registers for A_C are reused, while the 1 register for B_R has to load a new value from cache for

[1]This refers to the 128-bit SIMD&FP registers of Armv8-A, with space for 2 floating-point numbers [25].

each column of the register tile (N_R times). B_R resides in L1 cache as opposed to L2 cache, meaning repetitive loads are not as costly.

Enabling the micro-kernel increases performance to 38.6% of the theoretical machine peak, as seen in listing 5.6.

Listing 5.6: Executing the `hlo.matmul` benchmark after applying loop tiling with $M_C = 165$, $K_C = 720$, $M_R = 3$ and $N_R = 8$, explicit copying, and loop unrolling with $K_U = 1$.

```
$ mlir-opt -hlo-matmul-to-loops="tile=true tile-params=165,720,3,8 copy=true unroll=true
     unroll-factor=1" -convert-linalg-to-loops -lower-affine -convert-scf-to-cf -convert
    -cf-to-llvm -convert-arith-to-llvm -convert-func-to-llvm -expand-strided-metadata -
    finalize-memref-to-llvm="use-aligned-alloc=true" -reconcile-unrealized-casts -
    canonicalize input/hlo_matmul.mlir | mlir-cpu-runner -O3 -e main -entry-point-result
    =void -shared-libs=lib/libmlir_runner_utils.so -shared-libs=lib/
    libmlir_c_runner_utils.so
7.711173 GFLOPS
```

The unroll factor K_U

The downside of using loops is that they bring computational overhead for controlling the loop. The aim of unrolling the `kk`-loop by a factor K_U is to eliminate or reduce these instructions. This unroll factor corresponds to the step size of the loop and the number of times its loop-body is repeated. We introduce K_U as the unroll factor of the `kk`-loop, which is input alongside the `kk`-loop into `mlir::affine::loopUnrollJamByFactor(mlir::affine forOp, uint64_t unrollJamFactor)` in `HLOMatmulToLoops`. Note that the micro-kernel, as described in the previous section, corresponds to the `kk`-loop with $K_U = 1$ (no unrolling).

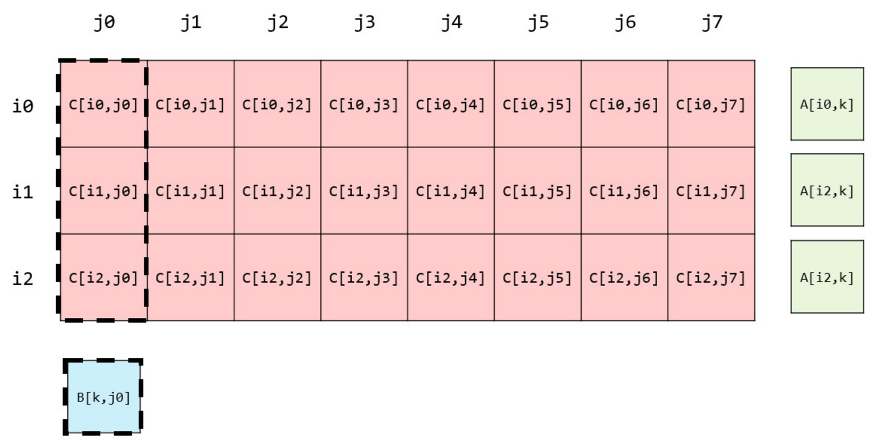
Naturally, the amount of code increases with K_U, meaning there is a space-time tradeoff to account for. For that reason, we are going to start off with a low value; $K_U = 2$. This increases performance to 47.5%, as seen in listing 5.7.

Listing 5.7: Executing the `hlo.matmul` benchmark after applying loop tiling with $M_C = 165$, $K_C = 720$, $M_R = 3$ and $N_R = 8$, explicit copying, and loop unrolling with $K_U = 2$.

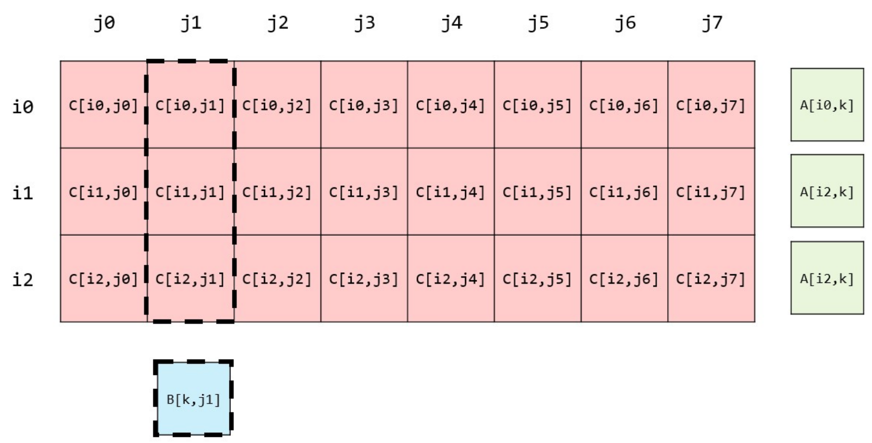
```
$ mlir-opt -hlo-matmul-to-loops="tile=true tile-params=165,720,3,8 copy=true unroll=true
     unroll-factor=2" -convert-linalg-to-loops -lower-affine -convert-scf-to-cf -convert
    -cf-to-llvm -convert-arith-to-llvm -convert-func-to-llvm -expand-strided-metadata -
    finalize-memref-to-llvm="use-aligned-alloc=true" -reconcile-unrealized-casts -
    canonicalize input/hlo_matmul.mlir | mlir-cpu-runner -O3 -e main -entry-point-result
    =void -shared-libs=lib/libmlir_runner_utils.so -shared-libs=lib/
    libmlir_c_runner_utils.so
9.496424 GFLOPS
```

5.3.4 Scalar Replacement

Post loop unrolling, there appears a number of redundant loads, meaning that the same values from A_C and B_R are loaded multiple times. As seen in listing 5.4, the load from A_C is invariant of `jjR`, meaning the unrolling of the `iiR`- and `jjR`-loops produce $N_R - 1$ redundant loads for each of the M_R loads from A_C. For B_R, $M_R - 1$ redundant loads for each of the N_R loads are produced. In other words, $M_R \cdot N_R$ loads are generated for both A_C



(a) For one column of a register tile of C, 3 values from A_C and 1 value from B_R are used.



(b) For the consecutive (and any other) column, the same 3 values of A_C are used while another value of B_R is used. The register used to store values from B_R and the resulting output column in the register tile are emphasized with dashed lines.

Figure 5.2: Illustration of a register tile of C (red) and additional registers reserved for A_C (green) and B_R (blue) with $N_R = 8$ and $M_R = 3$. The `i0`, `i1`, `j0`, `j1` and `k` labels are arbitrary index values.

and B_R, but only M_R and N_R are necessary for A_C and B_R, respectively. These issues are resolved by applying scalar replacement, which eliminates redundant loads and hoists invariant loads to an appropriate nest-level. We implement this as a last step post-canonicalization in `HLOMatmulToLoops::runOnOperation()` (see listing B.5 in the appendix) using `affine::affineScalarReplace`. As an example, listing C.10 in the appendix demonstrates the outcome of enabling scalar replacement with explicit copying disabled and $N_R = 2$ and $K_U = 1$ for readability. In there we can count 3 (= M_R) loads from A_C and 2 (= N_R) loads from B_R, meaning all redundant loads have been eliminated.

There is a slight improvement of performance when enabling scalar replacement, now at 47.6% of the theoretical machine peak, as seen in listing 5.8. This marginal increase in performance indicates that the LLVM middle- and/or back end already implement functionality to eliminate redundant loads.

Listing 5.8: Executing the `hlo.matmul` benchmark after applying loop tiling with $M_C = 165$, $K_C = 720$, $M_R = 3$ and $N_R = 8$, explicit copying, loop unrolling with $K_U = 2$, and scalar replacement.

```
$ mlir-opt -hlo-matmul-to-loops="tile=true tile-params=165,720,3,8 copy=true unroll=true
     unroll-factor=2 scalar-replace=true" -convert-linalg-to-loops -lower-affine -
    convert-scf-to-cf -convert-cf-to-llvm -convert-arith-to-llvm -convert-func-to-llvm -
    expand-strided-metadata -finalize-memref-to-llvm="use-aligned-alloc=true" -reconcile
    -unrealized-casts -canonicalize input/hlo_matmul.mlir | mlir-cpu-runner -O3 -e main
    -entry-point-result=void -shared-libs=lib/libmlir_runner_utils.so -shared-libs=lib/
    libmlir_c_runner_utils.so
9.529780 GFLOPS
```

5.3.5 Vectorization

The last optimization technique to apply is vectorization, which is the process of transforming scalar operations to operate on multiple data elements concurrently. In `hlo.matmul` (see listing C.6 in the appendix), the scalar operations in question are

```
%4 = arith.mulf %1, %2 fastmath<fast> : f64
%5 = arith.addf %3, %4 fastmath<fast> : f64
```

and vectorizing would, at the most basic level, mean to transform these into

```
%4 = arith.mulf %1, %2 fastmath<fast> : vector<2xf64>
%5 = arith.addf %3, %4 fastmath<fast> : vector<2xf64>
```

where the vectors are of length 2 to reflect the capacity of the vector registers [25].

We choose to follow the same approach as [5], where `%B` and `%C` are shape-cast into `memref<?x1024vector<2xf64>>` outside of the loop nest. There is no need to shape-cast `%A` since one value is used N_R times over, accounting for one row of the register tile (see figure 5.2). For `%A`, we are instead applying `vector.splat` after each load from `%A_C`, which produces a `vector` containing the given value. Said shape-cast operation does not already exist in MLIR, but Uday Bondhugula's implementation of it is available online [link]. For simplicity, we imported this implementation to our project as an operation within `mlir::memref`, with syntactical changes made where necessary.

Listing C.11 in the appendix presents an example where vectorization has been enabled. In addition to introducing `memref.shape_cast` and `vector.splat`, we have also replaced the lines

```
%7 = arith.mulf %4, %5 fastmath<fast> : vector<2xf64>
%8 = arith.addf %6, %7 fastmath<fast> : vector<2xf64>
```

with the line

```
%7 = vector.fma %4, %5, %6 : vector<2xf64>
```

which guarantees that the MLIR is lowered to the `llvm.fma.*` intrinsic [44]. This replacement does not necessarily have an effect on output, since fused multiply-add instructions may be generated by the LLVM backend regardless of whether we specify it in the MLIR or not. When enabling vectorization, there is a major improvement in performance, as seen in listing 5.9. With every optimization enabled, we reach 75.4% of the theoretical peak.

Listing 5.9: Executing the `hlo.matmul` benchmark after applying loop tiling with $M_C = 165$, $K_C = 720$, $M_R = 3$ and $N_R = 8$, explicit copying, loop unrolling with $K_U = 2$, scalar replacement and vectorization.

```
$ mlir—opt —hlo—matmul—to—loops="tile=true tile—params=165,720,3,8 copy=true unroll=true
     unroll—factor=2 scalar—replace=true vectorize=true" —convert—linalg—to—loops —lower
    —affine —convert—scf—to—cf —convert—cf—to—llvm —convert—arith—to—llvm —convert—func—
    to—llvm —expand—strided—metadata —finalize—memref—to—llvm="use—aligned—alloc=true" —
    reconcile—unrealized—casts —canonicalize input/hlo_matmul.mlir | mlir—cpu—runner —O3
     —e main —entry—point—result=void —shared—libs=lib/libmlir_runner_utils.so —shared—
    libs=lib/libmlir_c_runner_utils.so
15.074912 GFLOPS
```

5.3.6 Inspecting the Assembly

Since we were able to reach 93.0% of the theoretical peak with BLIS and ArmPL, we aim to improve the performance of `hlo.matmul` further. The limitations of the current parameter configuration ($M_C = 165$, $K_C = 720$, $M_R = 3$, $N_R = 8$ and $K_U = 2$) may become clear by inspecting the generated assembly. This will also let us observe whether we have managed to target vector instructions of Arm Neon. To simplify the analysis and minimize the size of listed code, we set $K_U = 1$, which corresponds to employing the original micro-kernel as described in section 5.3.3, for any discussion and analysis surrounding the generated assembly. Note that K_U modifies the `kk`-loop body and thus may affect the output assembly and furthermore register allocation. Listing 5.10 presents the output assembly, where we can confirm that the lower-level code has been vectorized as well.

Listing 5.10: Excerpt from output target assembly corresponding to the `kk`-loop body with $M_R = 3$, $N_R = 8$ and $K_U = 1$.

```
0000000000000000 <main>:
...
 4ec: 4ddfce94  ld1r {v20.2d}, [x20], #8
 4f0: ad7f5995  ldp q21, q22, [x12, #-32]
 4f4: f1000673  subs x19, x19, #0x1
 4f8: 4e74ced0  fmla v16.2d, v22.2d, v20.2d
 4fc: fd4b41b7  ldr d23, [x13, #5760]
 500: fd5681b8  ldr d24, [x13, #11520]
 504: 4fd712c7  fmla v7.2d, v22.2d, v23.d[0]
 508: 4fd812c6  fmla v6.2d, v22.2d, v24.d[0]
 50c: 4e74ceb3  fmla v19.2d, v21.2d, v20.2d
 510: 4fd712b2  fmla v18.2d, v21.2d, v23.d[0]
 514: 4fd812b1  fmla v17.2d, v21.2d, v24.d[0]
 518: acc25995  ldp q21, q22, [x12], #64
 51c: aa1403ed  mov x13, x20
 520: 4e74cec2  fmla v2.2d, v22.2d, v20.2d
 524: 4fd712c1  fmla v1.2d, v22.2d, v23.d[0]
 528: 4fd812c0  fmla v0.2d, v22.2d, v24.d[0]
 52c: 4e74cea5  fmla v5.2d, v21.2d, v20.2d
 530: 4fd712a4  fmla v4.2d, v21.2d, v23.d[0]
 534: 4fd812a3  fmla v3.2d, v21.2d, v24.d[0]
 538: 54fffda1  b.ne 4ec <main+0x4ec>  // b.any
...
```

There are multiple calls to

- `fmla vd, vn, vm` corresponding to `vmfaq_f64` [link], and
- `fmla vd, vn, vm[lane]` corresponding to either `vmfaq_lane_f64` [link] or `vmfaq_laneq_f64` [link].

These are vectorized floating-point fused multiply-add to accumulator intrinsics. The first operand is the destination (output) register and the two consecutive operands are the source (input) registers. Thus, the first operand corresponds to an element from C_R, and the second and third operands correspond to elements from A_C and B_R. When a lane is specified, it means the instruction is elementwise, only operating on the `lane`th index of the third operand. We will assume that the first source operand corresponds to B_R and the second A_C, since respective input sequence correspond to how the `jjR`- and `iiR`-loops have been unrolled. Then, we can also assume that the first line in listing 5.10,

```
4ec: 4ddfce94  ld1r {v20.2d}, [x20], #8
```

would stem from e.g.

```
%3 = affine.load %alloc[%arg3 * 3, %arg4 + %arg0 * 720] : memref<2088x2048xf64>
%4 = vector.splat %3 : vector<2xf64>
```

in the vectorized MLIR (see listing C.11 in the appendix), since `ld1r v20.2d, [x20], #8` [link] loads the value from register `x20` to the 2 64-bit lanes of `v20`. Note that the `d`- and `v`-registers we are referring to are interpretations of the 32 SIMD&FP registers; every SIMD&FP register occupy 128 bits, but the number of significant bits vary [25]. This means that, for example, `d0` occupies the same space in memory as the initial 64 bits of `v0`. The `d`-registers hold a single double-precision floating-point value, while the `v`-registers (otherwise called vector registers) may, for example, hold 2 double-precision floating-point elements [25]. How the 128 bits within a `v`-register is interpreted is determined by its suffix, `.2d` meaning 2 double-precision floating-point elements [25].

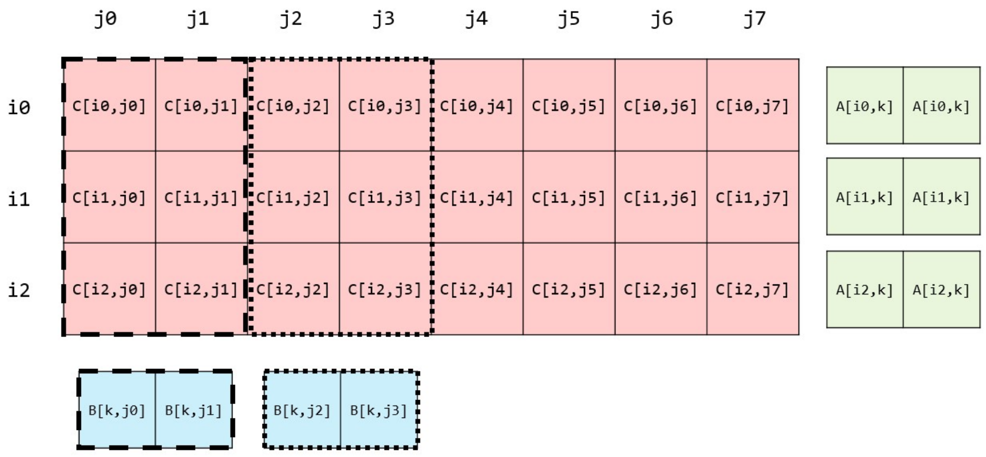


Figure 5.3: Illustration of a register tile of C (red) and additional registers reserved for A_C (green) and B_R (blue) with $N_R = 8, M_R = 3$. The first use of and resulting output from respective vector register used to store values from B_R are emphasized with dashed and dotted lines. `i0`-`i2`, `j0`-`j2` and `k` are arbitrary index values.

Judging from listing 5.10 and the inputs to the `fmla` instructions, `v0`-`v7` and `v16`-`v19` (12 in total) are used to store the output values, meaning they are reserved for the register tile of C. Then, `v21` and `v22` are used to store values from B_R, and `v20`, `v23` and `v24` are used to store values from A_C. From figure 5.2 we can reason that the minimum required number of registers is 16 when using $M_R = 3$ and $N_R = 8$; 12 vector registers for a register tile of C, one for each of the 3 values from A_C and 1 for every value of B_R. However, the total is instead 17, since 2 vector registers are used for B_R (see figure 5.3). Using 2 instead of 1 vector register for B_R means only $N_R/(2 \cdot 2) = 2$ calls have to be made to `ldp q21, q22 *` [link], which update the values inside both of the vectors. If 1 vector register had been used, $N_R/2 = 4$ calls would have to be made to e.g. `ldr q21 *` [link], which update the values inside a single vector.

5.3.7 Fine-Tuning the Parameters

Throughout section 5.3.6, it has been implied that $M_R = 3$ and $N_R = 8$ is a poor choice for register tiling, given that this configuration under-utilizes the 32 SIMD&FP registers. The question is whether it is possible to find a better configuration under the constraints of equations 5.1 and 5.2. From 5.1 with $M = 2088$ and $N = 2048$, we figure that M_R is a multiple of 2 and/or 3, and N_R is a multiple of 2. Ideally, assuming that it is possible and beneficial to use a single vector register to store values from B_R, we would find values that fulfill

$$M_R \cdot N_R/2 + M_R + 1 = 32 \tag{5.4}$$

but given the constraints, this is not mathematically possible. However, with $M_R = 3$ and $N_R = 16$ we expect to use 24 vector registers for output and 3 for values from A_C. Given that there are $32 - 24 - 3 = 5$ additional vector registers, and previously observed outcome (see figure 5.3), we can expect that at least 2 vector registers are going to be used to store values from B_R, making a total of 29 registers. Modifying all tile parameters accordingly, including adjusting M_C and K_C to retain the same sizes of A_C and B_R as seen in equation 5.3, proves to be beneficial. We reach a performance of 82.4% of the theoretical machine peak, as seen in listing 5.11.

Listing 5.11: Executing the `hlo.matmul` benchmark after applying loop tiling with $M_C = 330$, $K_C = 360$, $M_R = 3$ and $N_R = 16$, explicit copying, loop unrolling with $K_U = 2$, scalar replacement and vectorization.

```
$ mlir-opt -hlo-matmul-to-loops="tile=true tile-params=330,360,3,16 copy=true unroll=
    true unroll-factor=2 scalar-replace=true vectorize=true" -convert-linalg-to-loops -
    lower-affine -convert-scf-to-cf -convert-cf-to-llvm -convert-arith-to-llvm -convert-
    func-to-llvm -expand-strided-metadata -finalize-memref-to-llvm="use-aligned-alloc=
    true" -reconcile-unrealized-casts -canonicalize input/hlo_matmul.mlir | mlir-cpu-
    runner -O3 -e main -entry-point-result=void -shared-libs=lib/libmlir_runner_utils.so
     -shared-libs=lib/libmlir_c_runner_utils.so
16.474362 GFLOPS
```

Listing 5.12 presents the generated assembly after modifying the parameters to increase register use and setting $K_U = 1$, which ascertain that 29 registers are used in total in the micro-kernel. Again, Note that modyfing K_U may have an effect on output assembly and furthermore register allocation, and that $K_U = 1$ to simplify the analysis and minimize the size of listed code.

Listing 5.12: Excerpt from output target assembly corresponding to the `kk`-loop body with $M_R = 3$, $N_R = 16$ and $K_U = 1$.

```
580: 4ddfcde9  ld1r {v9.2d}, [x15], #8
584: ad7e2d8a  ldp q10, q11, [x12, #-64]
588: f10005ad  subs x13, x13, #0x1
58c: 4e69cd7c  fmla v28.2d, v11.2d, v9.2d
590: fd45a1cc  ldr d12, [x14, #2880]
594: fd4b41cd  ldr d13, [x14, #5760]
598: 4fcc117b  fmla v27.2d, v11.2d, v12.d[0]
59c: 4fcd117a  fmla v26.2d, v11.2d, v13.d[0]
5a0: 4e69cd5f  fmla v31.2d, v10.2d, v9.2d
5a4: 4fcc115e  fmla v30.2d, v10.2d, v12.d[0]
5a8: 4fcd115d  fmla v29.2d, v10.2d, v13.d[0]
5ac: ad7f2d8a  ldp q10, q11, [x12, #-32]
5b0: aa0f03ee  mov x14, x15
5b4: 4e69cd76  fmla v22.2d, v11.2d, v9.2d
5b8: 4fcc1175  fmla v21.2d, v11.2d, v12.d[0]
5bc: 4fcd1174  fmla v20.2d, v11.2d, v13.d[0]
5c0: 4e69cd59  fmla v25.2d, v10.2d, v9.2d
5c4: 4fcc1158  fmla v24.2d, v10.2d, v12.d[0]
5c8: 4fcd1157  fmla v23.2d, v10.2d, v13.d[0]
5cc: ad402d8a  ldp q10, q11, [x12]
5d0: 4e69cd70  fmla v16.2d, v11.2d, v9.2d
5d4: 4fcc1167  fmla v7.2d, v11.2d, v12.d[0]
5d8: 4fcd1166  fmla v6.2d, v11.2d, v13.d[0]
5dc: 4e69cd53  fmla v19.2d, v10.2d, v9.2d
5e0: 4fcc1152  fmla v18.2d, v10.2d, v12.d[0]
5e4: 4fcd1151  fmla v17.2d, v10.2d, v13.d[0]
5e8: ad412d8a  ldp q10, q11, [x12, #32]
5ec: 9102018c  add x12, x12, #0x80
5f0: 4e69cd62  fmla v2.2d, v11.2d, v9.2d
5f4: 4fcc1161  fmla v1.2d, v11.2d, v12.d[0]
5f8: 4fcd1160  fmla v0.2d, v11.2d, v13.d[0]
5fc: 4e69cd45  fmla v5.2d, v10.2d, v9.2d
600: 4fcc1144  fmla v4.2d, v10.2d, v12.d[0]
604: 4fcd1143  fmla v3.2d, v10.2d, v13.d[0]
608: 54fffbc1  b.ne 580 <main+0x580>  // b.any
```

Using the same reasoning as in section 5.3.6, we observe that `v0-v7` and `v16-v31` (24 in total) are reserved for the register tile, `v9`, `v12` and `v13` are used to store values from A_C, and `v10` and `v11` are used to store values from B_R. One could reason that $M_R = 6$ and $N_R = 8$ would be a preferable choice, given that this would be expected to use all 32 SIMD&FP registers if we assume that 6 and 2 vector registers are used to store values from A_C and B_R, respectively. We have, however, observed that using $M_R = 6$ and $N_R = 8$ produces output that does not follow the expected trend and yields poor performance. It is thus disregarded.

Furthermore, in addition to the advantages discussed in section 5.3.3, another advantage to unrolling the `kk`-loop is that it increases the number of SIMD&FP registers used in the micro-kernel. For $K_U = 8$, we observe that every register is used except for `v8`, as `v0-v7` and `v16-v31` are reserved for the register tile, `v9` is reserved for values from A_C and `v15` is reserved for values from B_R. The remaining 5 vector registers `v10-v14` are used for values from both A_C and B_R. Setting $K_U = 8$ leads to a performance at 85.6% of the theoretical peak, as seen in listing 5.13. We lose the intended benefit when increasing K_U any further. This could be due to the forementioned space-time tradeoff.

Listing 5.13: Executing the `hlo.matmul` benchmark after applying loop tiling with $M_C = 330$, $K_C = 360$, $M_R = 3$ and $N_R = 16$, explicit copying, loop unrolling with $K_U = 8$, scalar replacement and vectorization.

```
$ mlir-opt -hlo-matmul-to-loops="tile=true tile-params=330,360,3,16 copy=true unroll=
    true unroll-factor=8 vectorize=true" -convert-linalg-to-loops -lower-affine -convert
    -scf-to-cf -convert-cf-to-llvm -convert-arith-to-llvm -convert-func-to-llvm -expand-
    strided-metadata -finalize-memref-to-llvm="use-aligned-alloc=true" -reconcile-
    unrealized-casts -canonicalize input/hlo_matmul.mlir | mlir-cpu-runner -O3 -e main -
    entry-point-result=void -shared-libs=lib/libmlir_runner_utils.so -shared-libs=lib/
    libmlir_c_runner_utils.so
17.126675 GFLOPS
```

Ultimately, we have found that loop tiling, explicit copying, loop unrolling, scalar replacement and vectorization can be used to increase the performance of `hlo.matmul` from 1.9% to 85.6% — 8% from the performances achieved with ArmPL and BLIS. Table 5.1 summarizes the `hlo.matmul` benchmark results with respect to the parameter configurations tested. A dash (-) means that the (corresponding) optimization technique is disabled, and performance is presented in relation to the theoretical machine peak of 20 GFLOPS.

Table 5.1: Summary of results for the various parameter configurations tested in section 5.3.

M_C	K_C	M_R	N_R	Explicit copying	K_U	Scalar replacement	Vectorization	Performance [%]
-	-	-	-	-	-	-	-	1.9
165	720	3	8	-	-	-	-	9.7
165	720	3	8	yes	-	-	-	32.1
165	720	3	8	yes	1	-	-	38.6
165	720	3	8	yes	2	-	-	47.5
165	720	3	8	yes	2	yes	-	47.6
165	720	3	8	yes	2	yes	yes	75.4
330	360	3	16	yes	2	yes	yes	82.4
330	360	3	16	yes	8	yes	yes	85.6

Chapter 6
Reflections

In the pre-analysis, we found that the performance of the 2088×2048 double-precision matrix multiplication ranges from 0.4 GFLOPS for a naive-nest implementation compiled with `clang -O3 -ffast-math` to 18.55 GFLOPS for respective DGEMM routine of ArmPL and BLIS (see figure 6.1). This gap arises from the fact that high-performance compute libraries such as ArmPL and BLIS consist of hand-crafted routines specialized for certain purposes and hardware, unlike general-purpose compilers that leave the implementation and high-level optimizations to the end-user. Thus, implementing and compiling a generic 3-d loop nest, as we did in section 3.1, does not yield high performance. On the other hand, the advantage of using compilers is that the end-user is not bound to a finite set of low-level routines. A routine may yield excellent performance in isolated circumstances — as we have shown in this thesis for `cblas_dgemm` and `bli_dgemm` from ArmPL and BLIS, respectively — but then there is the question of how these routines should be incorporated to software applications, e.g. machine learning frameworks, as well as whether they can be utilized for different hardware platforms and architectures.

This brings us to MLIR-based code generation and the findings of this project. We were not able to prove that the MLIR-based DGEMM routine, `hlo.matmul`, could reach the performance of ArmPL and BLIS, but one must take into account that a limited set of configurations were tested (see table 5.1) due to time constraints. As seen in equation 4.1, the A_C and B_R buffers do not completely occupy the 1024 KiB available in L1 and L2 cache. We have not assessed whether M_C and K_C could be adjusted to account for that, under the constraints of equations 5.1 with $M_R = 3$ and 5.2. Furthermore, there exists other well-researched optimization strategies beside the ones chosen for this thesis that potentially could further improve performance. For one, when implementing loop tiling, we chose not to control memory allocation beyond the L1 and L2 cache levels. To consider the L3 cache level as well, we would have to make a partition along N with N_C and pack the resulting $K_C \times N_C$-sized tile B_C to a contiguous buffer [33].

However, we would argue that the difference of 8% between the performances of `hlo.matmul` and `bli_dgemm` of BLIS or `cblas_dgemm` of ArmPL is insignificant for our pur-

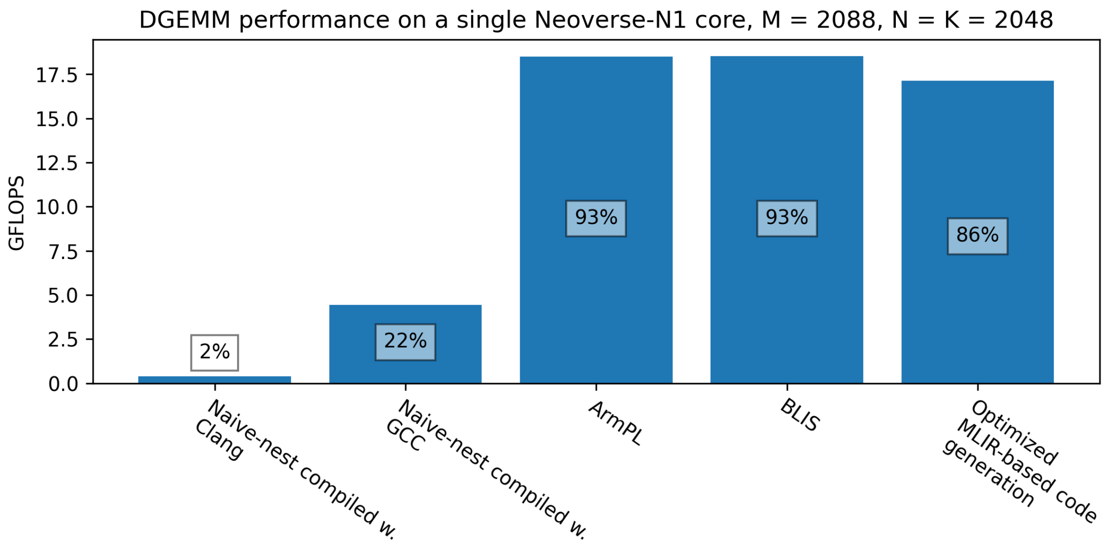


Figure 6.1: Overview of the benchmark results presented in this thesis. The height of each bar represents the absolute performance, while the overlaid percentage is the performance relative the theoretical machine peak of 20 GFLOPS.

poses, on the first hand being to showcase the possibility of implementing high-performance routines using an MLIR-based approach and on a higher level to present the MLIR framework as a reusable and extensible alternative to high-performance compute libraries common in machine learning frameworks. For research purposes, we restricted the experimentation to the Graviton2 CPU and as a natural consequence, our implementations likely will not work as expected for any other setup "out of the box". However, any necessary adjustment could be as simple as inputting an appropriate option value for the compiler pass. The loop tiling parameters, for example, can be adjusted for essentially any type of cache and register setup, assuming the chosen optimization strategy (see figure 5.1) is an appropriate choice. E.g., in sections 5.3.6 and 5.3.7, we showed how to use 17 as well as 29 registers in the micro-kernel while occupying the same amount of space in L1 and L2 cache by adjusting M_C, K_C, M_R and N_R. For any CPU that does not implement fused multiply-add vector instructions, the `vectorize` option should be set to `false`. It should also be noted that `-hlo-matmul-to-loops` could have been supplemented with more options beside the ones presented in this master's thesis, e.g., an option to control the vector length.

We believe the findings of this master's thesis will have a positive effect on society. MLIR is novel technology and may not be known by most computer scientists, or even compiler researchers and engineers. Given the discussed advantages of MLIR, we have a positive attitude toward MLIR being employed on a wider scale. The promising results of our experimentation as well as our demonstration of implementing an MLIR dialect and operation support that intention. This strengthens the MLIR project's aim of addressing software fragmentation and aiding in connecting existing compilers together.

6.1 Research Questions

RQ1 We showed that applying loop tiling, explicit copying, loop unrolling, scalar replacement and vectorization to an unoptimized GEMM operation (see listing C.6 in the appendix) increased performance from 1.7% to 85.6%. These optimization techniques were implemented within the `HLOMatmulToLoops` class using MLIR's pre-implemented resources, such as the `mlir::affine::tile` and `mlir::affine::interchangeLoops` functions which were used to achieve the general structure of the loop tiling scheme presented in listing 5.2.

RQ2 We vectorized `hlo.matmul` by shape-casting `%B` and `%C` into `memref<?x1024< 2xf64>>` and inserting the `vector.splat` and `vector.fma` operations into the body of the loop nest. In listing 5.10, we observed that these modifications were enough to target `vmfaq_f64` and `vmfaq_lane_f64` or `vmfaq_laneq_f64` — vector floating-point fused multiply-add to accumulator intrinsics of Arm Neon.

RQ3 When assessing performance as-is, `bli_dgemm` of BLIS and `cblas_dgemm` of ArmPL are superior to MLIR-based `hlo.matmul` by around 8%, when implementing the optimization strategy described in section 5.3. This is because such routines of high-performance compute libraries have been carefully hand-crafted for certain purposes and hardware. Still, we are of the opinion that we achieved high performance with MLIR as well. In terms of composability, we argue that MLIR is the better alternative. As was demonstrated in section 5.3, the MLIR framework consists of a wide array of pre-implemented resources that can be reused for different purposes. As opposed to typical high-performance compute libraries, the MLIR framework is not designated for a narrow range of hardware platforms. Yet, we were able to target vector machine instructions of Arm Neon and achieve high performance with MLIR. Given the options of `-hlo-matmul-to-loops` and the retargetability of the LLVM backend, we hypothesize that `hlo.matmul` could be used on, or adjusted for, various hardware and retain high performance.

6.2 Future Work

A natural, perhaps obvious, extension to this project would be to implement additional optimization techniques or expand on the techniques chosen for this master's thesis in order to bridge the gap between `hlo.matmul` and `cblas_dgemm` or `bli_dgemm`. Although, given the higher-level topic of this master's thesis being to improve machine learning systems from a software engineering perspective, we believe that it would have been more valuable to extend our efforts to building a (more or less) complete front end of a (machine learning) compiler and not restricting to the IR generation phase. In practice, this would mean to implement pre-processing, lexical analysis, syntax analysis and semantic analysis, generating an AST in likeness of figure 2.2, which would be further lowered to MLIR. This would elaborate on the topic of automatability, e.g. in automatically adapting the optimization parameters according to hardware-specific properties such as cache and register capacities and architecture, and further concretize the idea of leveraging MLIR-based code generation in a machine learning system. For the former, we could take inspiration from the machine description file of GCC, which specifies hardware-specific features and is used by the compiler to generate target code accordingly. Our version of the machine description file would consist of information relating to e.g. cache and registers. This information would be used to calculate appropriate values for M_C, K_C, M_R and N_R, using, for example, the proposed formulas of Low et. al. [33]. Additionally, the analysis could be extended to include standard routines or functions beside DGEMM, especially ones that make up typical "building blocks" within machine learning models.

Another aspect that would have been interesting to explore is higher-level optimization techniques that affect machine learning architecture on a larger scale, such as *operator fusion* — a key optimization in many state-of-the-art deep neural network execution frameworks, including TensorFlow [46]. Operator fusion is highly relevant to the objective of our research given that previous experiments in Apache SystemDS (previously Apache SystemML), an open source ML system for the end-to-end data science lifecycle [2], have shown up to 21x performance improvements with optimized fusion plans compared to hand-written fused operators, with negligible optimization and code generation overhead [4].

Since there already exist projects leveraging MLIR for machine learning software, such as OpenXLA [50], an alternative approach to our research objective would be to piggyback off of these infrastructures, instead of constructing a framework from scratch.

6.3 Conclusion

In this master's thesis, we have demonstrated the implementation of an optimized double-precision GEMM routine using an MLIR-based approach, yielding a performance only 8% from the effective machine peak on a Neoverse-N1 core.

Using MLIR resources, we were able to efficiently implement a compiler pass lowering the high-level `hlo.matmul` operation to a semantically equivalent loop nest, equipped with options enabling control over the optimization process and furthermore the design of a suitable micro-kernel. Despite optimizing on high-level IR, we were able to target the vectorized floating-point fused multiply-add to accumulator intrinsics of Neon and control register allocation. This is indicative of the potential benefits of incorporating MLIR in the infrastructure of machine learning systems, in constituting retargetability while retaining high performance.

References

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Aug. 2006. ISBN: 0321486811. URL: `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20%5C&amp;path=ASIN/0321486811`.

[2] Apache. *Apache SystemDS™*. URL: `https://systemds.apache.org/` (visited on 12/19/2023).

[3] AWS. *AWS Graviton Technical Guide*. URL: `https://github.com/aws/aws-graviton-getting-started/tree/main#readme` (visited on 11/15/2023).

[4] Matthias Boehm et al. "On optimizing operator fusion plans for large-scale machine learning in systemML". In: *Proc. VLDB Endow.* 11.12 (Aug. 2018), pp. 1755–1768. ISSN: 2150-8097. DOI: `10.14778/3229863.3229865`. URL: `https://doi.org/10.14778/3229863.3229865`.

[5] Uday Bondhugula. *High Performance Code Generation in MLIR: An Early Case Study with GEMM*. 2020. arXiv: `2003.00532 [cs.PF]`.

[6] Tianqi Chen et al. "TVM: an automated end-to-end optimizing compiler for deep learning". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594. ISBN: 9781931971478.

[7] Clang. *clang - the Clang C, C++, and Objective-C compiler*. URL: `https://clang.llvm.org/docs/CommandGuide/clang.html` (visited on 11/15/2023).

[8] Clang. *Clang Compiler User's Manual*. URL: `https://clang.llvm.org/docs/UsersManual.html` (visited on 11/15/2023).

[9] Clang. *Getting Started: Building and Running Clang*. URL: `https://clang.llvm.org/get_started.html` (visited on 11/15/2023).

[10] Quentin Colombet. *[mlir][Conversion] Rename the MemRefToLLVM pass*. URL: `https://reviews.llvm.org/D142463` (visited on 01/10/2024).

[11] Quentin Colombet. *PSA: You need to run 'expand-strided-metadata' before 'memref-to-llvm' now.* URL: `https://discourse.llvm.org/t/psa-you-need-to-run-expand-strided-metadata-before-memref-to-llvm-now/66956` (visited on 01/10/2024).

[12] Keith D. Cooper and Linda Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2022. ISBN: 9780128189269. URL: `https://www.amazon.com/Engineering-Compiler-Keith-D-Cooper/dp/0128154128`.

[13] Intel Corporation. *Intel oneDNN AI Optimizations Enabled as Default in TensorFlow.* URL: `https://www.intel.com/content/www/us/en/newsroom/news/intel-onednn-speeds-ai-optimizations-in-tensorflow.html` (visited on 11/21/2023).

[14] Intel Corporation. *Intel® oneAPI Deep Neural Network Developer Guide and Reference.* URL: `https://www.intel.com/content/www/us/en/docs/onednn/developer-guide-reference/2023-1/build-options.html` (visited on 11/21/2023).

[15] Intel Corporation. *Intel® oneAPI Math Kernel Library.* URL: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html` (visited on 11/27/2023).

[16] Deci. *An Introduction to the Inference Stack and Inference Acceleration Techniques.* URL: `https://deci.ai/blog/inference-stack-and-inference-acceleration-techniques/#/` (visited on 12/14/2023).

[17] Ray Fernandez. *Google Gemini AI: Everything We Know So Far.* URL: `https://www.techopedia.com/everything-we-know-about-google-gemini` (visited on 12/14/2023).

[18] GCC. *Options That Control Optimization.* URL: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` (visited on 11/15/2023).

[19] Robert van de Geijn and Kazushige Goto. "Anatomy of high-performance matrix multiplication Kazushige Goto, Robert A. van de Geijn ACM Transactions on Mathematical Software (TOMS), 2008". In: *ACM Transactions on Mathematical Software* 34 (May 2008), Article 12. DOI: `10.1145/1356052.1356053`.

[20] Science of High Performance Computing group. URL: `https://github.com/flame/blis/blob/master/docs/BuildSystem.md` (visited on 11/15/2023).

[21] IREE. *IREE.* URL: `https://iree.dev/` (visited on 12/15/2023).

[22] Kaggle. *State of Data Science and Machine Learning 2022.* URL: `https://www.kaggle.com/kaggle-survey-2022` (visited on 12/11/2023).

[23] Chris Lattner and Tim Davis. *MLIR: accelerating AI with open-source infrastructure.* Sept. 2019. URL: `https://blog.google/technology/ai/mlir-accelerating-ai-open-source-infrastructure/`.

[24] Chris Lattner et al. *MLIR: A Compiler Infrastructure for the End of Moore's Law.* 2020. arXiv: `2002.11054 [cs.PL]`.

[25] Arm Limited. *Arm Architecture Reference Manual. for A-profile architecture.*

[26] Arm Limited. *Arm Compute Library.* URL: `https://www.arm.com/technologies/compute-library` (visited on 12/15/2023).

[27] Arm Limited. *Arm Performance Libraries*. URL: `https://developer.arm.com/Tools%20and%20Software/Arm%20Performance%20Libraries` (visited on 12/15/2023).

[28] Arm Limited. *Get started with Arm Performance Libraries (standalone version)*. URL: `https://developer.arm.com/documentation/109408/0100/Installation/Install-on-Linux?lang=en` (visited on 11/15/2023).

[29] LLVM. *Clang: a C language family frontend for LLVM*. URL: `https://clang.llvm.org/` (visited on 11/13/2023).

[30] LLVM. *Getting Started with the LLVM System*. URL: `https://llvm.org/docs/GettingStarted.html#requirements` (visited on 11/15/2023).

[31] LLVM. *The LLVM Compiler Infrastructure*. URL: `https://llvm.org/` (visited on 11/13/2023).

[32] LLVM. *Writing an LLVM Pass*. URL: `https://llvm.org/docs/WritingAnLLVMPass.html` (visited on 11/27/2023).

[33] Tze Meng Low et al. "Analytical Modeling Is Enough for High-Performance BLIS". In: *ACM Trans. Math. Softw.* 43.2 (Aug. 2016). ISSN: 0098-3500. DOI: `10.1145/2925987`. URL: `https://doi.org/10.1145/2925987`.

[34] MLIR. *Chapter 6: Lowering to LLVM and CodeGeneration*. URL: `https://mlir.llvm.org/docs/Tutorials/Toy/Ch-6/` (visited on 11/14/2023).

[35] MLIR. *Creating a Dialect*. URL: `https://mlir.llvm.org/docs/Tutorials/CreatingADialect/` (visited on 11/23/2023).

[36] MLIR. *Getting Started*. URL: `https://mlir.llvm.org/getting_started/` (visited on 11/15/2023).

[37] MLIR. *Interfaces*. URL: `https://mlir.llvm.org/docs/Interfaces/3` (visited on 01/11/2024).

[38] MLIR. *MLIR Language Reference*. URL: `https://mlir.llvm.org/docs/LangRef/` (visited on 11/14/2023).

[39] MLIR. *MLIR Rationale*. URL: `https://mlir.llvm.org/docs/Rationale/Rationale/#` (visited on 12/21/2023).

[40] MLIR. *Multi-Level Intermediate Representation Overview*. 2023. URL: `https://mlir.llvm.org/` (visited on 11/27/2023).

[41] MLIR. *Op vs. Operation: Using MLIR Operations*. URL: `https://mlir.llvm.org/docs/Tutorials/Toy/Ch-2/#op-vs-operation-using-mlir-operations` (visited on 11/14/2023).

[42] MLIR. *Pass Infrastructure*. URL: `https://mlir.llvm.org/docs/PassManagement/` (visited on 11/14/2023).

[43] MLIR. *Users of MLIR*. 2023. URL: `https://mlir.llvm.org/users/` (visited on 01/19/2024).

[44] MLIR. *`vector.fma` (vector::FMAOp)*. URL: `https://mlir.llvm.org/docs/Dialects/Vector/#vectorfma-vectorfmaop` (visited on 01/20/2024).

[45] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN: 1558603204.

[46] Wei Niu et al. "DNNFusion: accelerating deep neural networks execution with advanced operator fusion". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 883–898. ISBN: 9781450383912. DOI: `10.1145/3453483.3454083`. URL: `https://doi.org/10.1145/3453483.3454083`.

[47] Diego Novillo. "GCC—An Architectural Overview, Current Status, and Future Directions". In: 2010. URL: `https://api.semanticscholar.org/CorpusID:49334934`.

[48] OpenXLA. *StableHLO*. URL: `https://github.com/openxla/stablehlo#stablehlo` (visited on 01/19/2024).

[49] Nadav Rotem et al. *Glow: Graph Lowering Compiler Techniques for Neural Networks*. 2019. arXiv: `1805.00907 [cs.PL]`.

[50] James Rubin. *OpenXLA is available now to accelerate and simplify machine learning*. Mar. 2023. URL: `https://opensource.googleblog.com/2023/03/openxla-is-ready-to-accelerate-and-simplify-ml-development.html`.

[51] Rust. *Rust Compiler Development Guide*. URL: `https://rustc-dev-guide.rust-lang.org/overview.html#intermediate-representations` (visited on 11/13/2023).

[52] Maximilian Schreiner. *GPT-4 architecture, datasets, costs and more leaked*. URL: `https://the-decoder.com/gpt-4-architecture-datasets-costs-and-more-leaked/` (visited on 12/14/2023).

[53] Amazon Web Services. *Amazon EC2 C6g Instances*. URL: `https://aws.amazon.com/ec2/instance-types/c6g/` (visited on 11/27/2023).

[54] Amazon Web Services. *Instance Type Details*. URL: `https://aws.amazon.com/ec2/instance-types/` (visited on 11/27/2023).

[55] Amazon Web Services. *Optimize CPU options*. URL: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-optimize-cpu.html` (visited on 11/27/2023).

[56] The GCC Team. *GCC, the GNU Compiler Collection*. URL: `https://gcc.gnu.org/` (visited on 12/20/2023).

[57] The TensorFlow MLIR Team. *MLIR: A new intermediate representation and compiler framework*. Apr. 2019. URL: `https://medium.com/tensorflow/mlir-a-new-intermediate-representation-and-compiler-framework-beba999ed18d`.

[58] TensorFlow. *What's new in TensorFlow 2.10?* 2022. URL: `https://blog.tensorflow.org/2022/09/whats-new-in-tensorflow-210.html` (visited on 08/01/2023).

[59] TensorFlow. *XLA architecture*. URL: `https://www.tensorflow.org/xla/architecture` (visited on 12/15/2023).

[60] Field G. Van Zee and Robert A. van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Trans. Math. Softw.* 41.3 (June 2015). ISSN: 0098-3500. DOI: `10.1145/2764454`. URL: `https://doi.org/10.1145/2764454`.

[61] Nicolas Vasilache and Javier Setoain. *[RFC] Vector Dialects: Neon and SVE*. 2020. URL: `https://discourse.llvm.org/t/rfc-vector-dialects-neon-and-sve/2284` (visited on 11/20/2023).

[62] Nicolas Vasilache et al. *Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction*. 2022. arXiv: `2202.03293 [cs.PL]`.

[63] Alex Zinenko. *PSA: run -reconcile-unrealized-casts after all -convert-*-to-llvm from now on*. URL: `https://discourse.llvm.org/t/psa-run-reconcile-unrealized-casts-after-all-convert-to-llvm-from-now-on/4266` (visited on 01/10/2024).

Appendices

Appendix A
Benchmark Programs

A.1 MLIR

The program presented in listing A.1 was used for benchmarking in chapter 4, while the program presented in listing A.2 was used in chapter 5. Both benchmark programs are based on the `mlir/benchmark/dgemm-hop.mlir` file from the MLIRX repository[1].

Listing A.1: Benchmark of naive-nest implementation of equation 1.1, with $M = 2088$ and $N = K = 2048$.

```
func.func @matmul(%A: memref<2088x2048xf64>, %B: memref<2048x2048xf64>, %C: memref
    <2088x2048xf64>) {
    affine.for %arg3 = 0 to 2088 {
        affine.for %arg4 = 0 to 2048 {
            affine.for %arg5 = 0 to 2048 {
                %a = affine.load %A[%arg3, %arg5] : memref<2088x2048xf64>
                %b = affine.load %B[%arg5, %arg4] : memref<2048x2048xf64>
                %ci = affine.load %C[%arg3, %arg4] : memref<2088x2048xf64>
                %p = arith.mulf %a, %b : f64
                %co = arith.addf %ci, %p : f64
                affine.store %co, %C[%arg3, %arg4] : memref<2088x2048xf64>
            }
        }
    }
return
}

func.func @main() {
    %reps = index.constant 5

    %A = memref.alloc() : memref<2088x2048xf64>
    %B = memref.alloc() : memref<2048x2048xf64>
    %C = memref.alloc() : memref<2088x2048xf64>
    %cf1 = llvm.mlir.constant(1.00000e+00 : f64) : f64

    linalg.fill ins(%cf1 : f64) outs(%A : memref<2088x2048xf64>)
```

[1]MLIRX was MLIR with extensions, although defunct since 2 December 2023. The original benchmark program can be viewed by reverting to commit with hash 120a9c93a3286c6745f3a824dd73521cd7a18dab [link].

```
    linalg.fill ins(%cf1 : f64) outs(%B : memref<2048x2048xf64>)

    %t_start = call @rtclock() : () -> (f64)
    affine.for %ti = 0 to %reps {
      linalg.fill ins(%cf1 : f64) outs(%C : memref<2088x2048xf64>)
      func.call @matmul(%A, %B, %C) : (memref<2088x2048xf64>, memref<2048x2048xf64
    >, memref<2088x2048xf64>) -> ()
    }
    %t_end = call @rtclock() : () -> (f64)

    %c0 = index.constant 0
    %c1 = index.constant 1
    %M = memref.dim %C, %c0 : memref<2088x2048xf64>
    %N = memref.dim %C, %c1 : memref<2088x2048xf64>
    %K = memref.dim %A, %c1 : memref<2088x2048xf64>

    %t = arith.subf %t_end, %t_start : f64
    %f1 = arith.muli %M, %N : index
    %f2 = arith.muli %f1, %K : index
    // 2*M*N*K.
    %c2 = index.constant 2
    %f3 = arith.muli %c2, %f2 : index
    %num_flops = arith.muli %reps, %f3 : index
    %num_flops_i = arith.index_cast %num_flops : index to i64
    %num_flops_f = arith.sitofp %num_flops_i : i64 to f64
    %flops = arith.divf %num_flops_f, %t : f64

    call @printF64(%t) : (f64) -> ()
    call @printNewline() : () -> ()
    call @printFlops(%flops) : (f64) -> ()

    memref.dealloc %A : memref<2088x2048xf64>
    memref.dealloc %B : memref<2048x2048xf64>
    memref.dealloc %C : memref<2088x2048xf64>

    return
}

func.func private @printNewline() -> ()
func.func private @printF64(f64) -> ()
func.func private @printFlops(f64) -> ()
func.func private @rtclock() -> (f64)
```

Listing A.2: Benchmark of `hlo.matmul`, with $M = 2088$ and $N = K = 2048$.

```
func.func @main() {
    %reps = index.constant 5

    %A = memref.alloc() : memref<2088x2048xf64>
    %B = memref.alloc() : memref<2048x2048xf64>
    %C = memref.alloc() : memref<2088x2048xf64>
    %cf1 = llvm.mlir.constant(1.00000e+00 : f64) : f64

    linalg.fill ins(%cf1 : f64) outs(%A : memref<2088x2048xf64>)
    linalg.fill ins(%cf1 : f64) outs(%B : memref<2048x2048xf64>)

    %t_start = call @rtclock() : () -> (f64)
    affine.for %ti = 0 to %reps {
      linalg.fill ins(%cf1 : f64) outs(%C : memref<2088x2048xf64>)
      hlo.matmul ins(%A : memref<2088x2048xf64>, %B : memref<2048x2048xf64>, %C :
    memref<2088x2048xf64>)
    }
    %t_end = call @rtclock() : () -> (f64)

    %c0 = index.constant 0
    %c1 = index.constant 1
    %M = memref.dim %C, %c0 : memref<2088x2048xf64>
    %N = memref.dim %C, %c1 : memref<2088x2048xf64>
    %K = memref.dim %A, %c1 : memref<2088x2048xf64>

    %t = arith.subf %t_end, %t_start : f64
    %f1 = arith.muli %M, %N : index
    %f2 = arith.muli %f1, %K : index
    // 2*M*N*K.
    %c2 = index.constant 2
    %f3 = arith.muli %c2, %f2 : index
    %num_flops = arith.muli %reps, %f3 : index
    %num_flops_i = arith.index_cast %num_flops : index to i64
    %num_flops_f = arith.sitofp %num_flops_i : i64 to f64
    %flops = arith.divf %num_flops_f, %t : f64

    call @printF64(%t) : (f64) -> ()
    call @printNewline() : () -> ()
    call @printFlops(%flops) : (f64) -> ()

    memref.dealloc %A : memref<2088x2048xf64>
    memref.dealloc %B : memref<2048x2048xf64>
    memref.dealloc %C : memref<2088x2048xf64>

    return
}

func.func private @printNewline() -> ()
func.func private @printF64(f64) -> ()
func.func private @printFlops(f64) -> ()
func.func private @rtclock() -> (f64)
```

Appendix B
Source files

B.1 MLIR

B.1.1 Toy Tutorial

MLIR's *Toy Tutorial* walks through the implementation of a basic MLIR-based programming language. Listing B.1 presents an example of how to emit LLVM IR from and MLIR module.

Listing B.1: Function exporting an MLIR module into LLVM IR. Taken from [34].

```
int dumpLLVMIR(mlir::ModuleOp module) {
  // Translate the module, that contains the LLVM dialect, to LLVM IR. Use a
  // fresh LLVM IR context. (Note that LLVM is not thread-safe and any
  // concurrent use of a context requires external locking.)
  llvm::LLVMContext llvmContext;
  auto llvmModule = mlir::translateModuleToLLVMIR(module, llvmContext);
  if (!llvmModule) {
    llvm::errs() << "Failed to emit LLVM IR\n";
    return -1;
  }

  // Initialize LLVM targets.
  llvm::InitializeNativeTarget();
  llvm::InitializeNativeTargetAsmPrinter();
  mlir::ExecutionEngine::setupTargetTriple(llvmModule.get());

  /// Optionally run an optimization pipeline over the llvm module.
  auto optPipeline = mlir::makeOptimizingTransformer(
      /*optLevel=*/EnableOpt ? 3 : 0, /*sizeLevel=*/0,
      /*targetMachine=*/nullptr);
  if (auto err = optPipeline(llvmModule.get())) {
    llvm::errs() << "Failed to optimize LLVM IR " << err << "\n";
    return -1;
  }
  llvm::errs() << *llvmModule << "\n";
  return 0;
}
```

B.1.2 HLO dialect

The directory tree below displays all the files needed to implement `hlo`.

```
llvm-project
└─mlir
  ├─include
  │ └─mlir
  │   └─Dialect
  │     └─HLO
  │       ├─IR
  │       │ ├─CMakeLists.txt
  │       │ ├─HLO.h
  │       │ ├─HLOBase.td
  │       │ └─HLOOps.td
  │       ├─Transforms
  │       │ ├─CMakeLists.txt
  │       │ ├─Passes.h
  │       │ └─Passes.td
  │       └─CMakeLists.txt
  └─lib
    └─Dialect
      └─HLO
        ├─IR
        │ ├─CMakeLists.txt
        │ ├─HLODialect.cpp
        │ └─HLOOps.cpp
        ├─Transforms
        │ ├─CMakeLists.txt
        │ └─HLOMatmulToLoops.cpp
        └─CMakeLists.txt
```

In general, the `include` directory is reserved for public header files and `lib` for sources [35]. We have chosen to reserve the `IR` subdirectories for basic declarations and definitions pertaining to `hlo` and `hlo.matmul`, and `Transforms` for the `-hlo-matmul-to-loops` pass.

Listings B.2, B.3 and B.4 present the `.td` files that are used to TableGen'erate boilerplate code.

Listing B.2: TableGen file `HLOBase.td`.

```
include "mlir/IR/DialectBase.td"

def HLO_Dialect : Dialect {
  let name = "hlo";

  let summary = "A minimal dialect consisting of High-Level linear algebra
    Operations.";

  let description = [{
    HLO is a minimal dialect consisting of High-Level linear algebra Operations.
  }];

  let cppNamespace = "::mlir::hlo";

  let dependentDialects = [
    "affine::AffineDialect",
    "arith::ArithDialect",
    "memref::MemRefDialect",
    "vector::VectorDialect"
  ];

  let extraClassDeclaration = [{
    /// Register all dialect operations.
    void registerOperations();
  }];
}
```

Listing B.3: TableGen file `HLOOps.td`.

```
include "mlir/IR/OpBase.td"
include "mlir/Dialect/HLO/IR/HLOBase.td"

#ifndef HLO_OPS
#define HLO_OPS

class HLO_Op<string mnemonic, list<Trait> traits = []> :
  Op<HLO_Dialect, mnemonic, traits> {}

def MatmulOp : HLO_Op<"matmul"> {
  let arguments = (ins F64MemRef:$A, F64MemRef:$B, F64MemRef:$C);
  let assemblyFormat = [{
    attr-dict
    `ins` `(` $A `:` type($A) `,` $B `:` type($B) `,` $C `:` type($C) `)`
  }];
}

#endif // HLO_OPS
```

Listing B.4: TableGen file `Passes.td`.

```
#ifndef MLIR_DIALECT_HLO_PASSES
#define MLIR_DIALECT_HLO_PASSES

include "mlir/Pass/PassBase.td"

def HLOMatmulToLoops : Pass<"hlo-matmul-to-loops", "func::FuncOp"> {
  let summary = "...";
  let constructor = "mlir::hlo::createHLOMatmulToLoopsPass()";
  let dependentDialects = ["affine::AffineDialect", "arith::ArithDialect", "memref
    ::MemRefDialect", "vector::VectorDialect"];
  let options = [
    Option<"enableTiling", "tile", "bool",
            /*default=*/ "false",
            "Enables loop tiling.">,
    ListOption<"tileParameters", "tile-params", "unsigned",
            "Loop tile parameters M_C, K_C, M_R and N_R.">,
    Option<"enableExplicitCopying", "copy", "bool",
            /*default=*/ "false",
            "Enables explicit copying.">,
    Option<"enableUnrolling", "unroll", "bool",
            /*default=*/ "false",
            "Enables loop unrolling.">,
    Option<"unrollFactor", "unroll-factor", "unsigned",
            /*default=*/ "4",
            "Unroll factor for the k-loop.">,
    Option<"enableScalarReplacement", "scalar-replace", "bool",
            /*default=*/ "false",
            "Enables vectorization.">,
    Option<"enableVectorization", "vectorize", "bool",
            /*default=*/ "false",
            "Enables vectorization.">,
  ];
}

#endif // MLIR_DIALECT_HLO_PASSES
```

The majority of the implementation resides in `HLOMatmulToLoops.cpp`, partially presented in listing B.5.

Listing B.5: Abbreviated version of `HLOMatmulToLoops.cpp`

```
/* included files */

namespace mlir {
namespace hlo {
#define GEN_PASS_DEF_HLOMATMULTOLOOPS
#include "mlir/Dialect/HLO/Transforms/Passes.h.inc"
} // namespace hlo
} // namespace mlir

namespace {

struct HLOMatmulToLoops
    : public hlo::impl::HLOMatmulToLoopsBase<
          HLOMatmulToLoops> {
  void runOnOperation() override;
  /* extra implementation-specific definitions */
};

} // namespace

std::unique_ptr<OperationPass<func::FuncOp>>
mlir::hlo::createHLOMatmulToLoopsPass() {
  return std::make_unique<HLOMatmulToLoops>();
}

/* helper functions to construct the new operation */

class HLOMatmulRewritePattern : public OpRewritePattern<hlo::MatmulOp> {
public:
  HLOMatmulRewritePattern(MLIRContext *context, bool enableTiling, ArrayRef<
    unsigned> tileParams, bool enableExplicitCopying, bool enableUnrolling,
    unsigned unrollFactor, bool enableVectorization) :
                          OpRewritePattern<hlo::MatmulOp>(context, 1),
                          enableTiling(enableTiling), tileParams(tileParams),
                          enableExplicitCopying(enableExplicitCopying),
                          enableUnrolling(enableUnrolling), unrollFactor(
    unrollFactor),
                          enableVectorization(enableVectorization) {}

  LogicalResult matchAndRewrite(hlo::MatmulOp op,
                                PatternRewriter &rewriter) const override {
    /* construct a new operation */
    rewriter.replaceOp(op, newOp);
  }

private:
  bool enableTiling;
  ArrayRef<unsigned> tileParams;
  bool enableExplicitCopying;
  bool enableUnrolling;
  unsigned unrollFactor;
  bool enableVectorization;
};


void HLOMatmulToLoops::runOnOperation() {
  auto enclosingOp = getOperation();
  MLIRContext *context = enclosingOp.getContext();
  RewritePatternSet patterns(context);
  /* ... */
```

```
patterns.add<HLOMatmulRewritePattern>(context, enableTiling, tileParams,
  enableExplicitCopying, enableUnrolling, unrollFactor, enableVectorization);
/* additional rewrite patterns for canonicalization */
applyPatternsAndFoldGreedily(enclosingOp, std::move(patterns));
/* ... */
```

Appendix C

Demonstrations

C.1 Emitting LLVM IR with Clang

Using the `-emit-llvm` flag, we can observe the LLVM IR emitted from a file input to `clang`. Listing C.2 presents the LLVM IR generated from the input file `multiply.c` (see listing C.1).

Listing C.1: Sample input file `multiply.c` consisting of a simple multiplication function.

```c
int multiply() {
    int a = 4;
    int b = 10;
    int c = a * b;

    return c;
}
```

Listing C.2: Produced output file `multiply.ll` from running `clang -S -emit-llvm multiply.c`.

```
; ModuleID = 'c/multiply.c'
source_filename = "c/multiply.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
    :16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @multiply() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 4, i32* %1, align 4
  store i32 10, i32* %2, align 4
  %4 = load i32, i32* %1, align 4
  %5 = load i32, i32* %2, align 4
  %6 = mul nsw i32 %4, %5
  store i32 %6, i32* %3, align 4
  %7 = load i32, i32* %3, align 4
  ret i32 %7
}
```

```

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
```

C.2 The `-affine-*` passes

Listing C.3 presents `@matmul` (see listing A.1) transformed with MLIR's pre-implemented passes.

Listing C.3: `@matmul` with loop tiling and explicit copying applied, using the `-affine-loop-tile` and `-affine-data-copy-generate` passes.

```
func.func @matmul(%arg0: memref<2088x2048xf64>, %arg1: memref<2048x2048xf64>, %arg2: memref<2088x2048xf64>) {
  %c118784 = arith.constant 118784 : index
  %c0 = arith.constant 0 : index
  %c118784_0 = arith.constant 118784 : index
  %c0_1 = arith.constant 0 : index
  %c4096 = arith.constant 4096 : index
  %c0_2 = arith.constant 0 : index
  %c1856 = arith.constant 1856 : index
  %c0_3 = arith.constant 0 : index
  %c0_4 = arith.constant 0 : index
  affine.for %arg3 = 0 to 2088 step 232 {
    affine.for %arg4 = 0 to 2048 step 512 {
      %0 = affine.apply affine_map<(d0, d1) -> (d0)>(%arg3, %arg4)
      %1 = affine.apply affine_map<(d0, d1) -> (d1)>(%arg3, %arg4)
      %alloc = memref.alloc() : memref<232x512xf64>
      affine.for %arg5 = affine_map<(d0) -> (d0)>(%arg3) to affine_map<(d0) -> (d0 + 232)>(%arg3) {
        affine.for %arg6 = affine_map<(d0) -> (d0)>(%arg4) to affine_map<(d0) -> (d0 + 512)>(%arg4) {
          %4 = affine.load %arg2[%arg5, %arg6] : memref<2088x2048xf64>
          affine.store %4, %alloc[%arg5 - %arg3, %arg6 - %arg4] : memref<232x512xf64>
        }
      }
      affine.for %arg5 = 0 to 2048 step 8 {
        %4 = affine.apply affine_map<(d0, d1) -> (d0)>(%arg3, %arg5)
        %5 = affine.apply affine_map<(d0, d1) -> (d1)>(%arg3, %arg5)
        %alloc_5 = memref.alloc() : memref<232x8xf64>
        affine.for %arg6 = affine_map<(d0) -> (d0)>(%arg3) to affine_map<(d0) -> (d0 + 232)>(%arg3) {
          affine.for %arg7 = affine_map<(d0) -> (d0)>(%arg5) to affine_map<(d0) -> (d0 + 8)>(%arg5) {
            %8 = affine.load %arg0[%arg6, %arg7] : memref<2088x2048xf64>
            affine.store %8, %alloc_5[%arg6 - %arg3, %arg7 - %arg5] : memref<232x8xf64>
          }
        }
        %6 = affine.apply affine_map<(d0, d1) -> (d0)>(%arg5, %arg4)
        %7 = affine.apply affine_map<(d0, d1) -> (d1)>(%arg5, %arg4)
        %alloc_6 = memref.alloc() : memref<8x512xf64>
        affine.for %arg6 = affine_map<(d0) -> (d0)>(%arg5) to affine_map<(d0) -> (d0 + 8)>(%arg5) {
          affine.for %arg7 = affine_map<(d0) -> (d0)>(%arg4) to affine_map<(d0) -> (d0 + 512)>(%arg4) {
            %8 = affine.load %arg1[%arg6, %arg7] : memref<2048x2048xf64>
            affine.store %8, %alloc_6[%arg6 - %arg5, %arg7 - %arg4] : memref<8x512xf64>
          }
        }
        affine.for %arg6 = affine_map<(d0) -> (d0)>(%arg3) to affine_map<(d0) -> (d0 + 232)>(%arg3) {
          affine.for %arg7 = affine_map<(d0) -> (d0)>(%arg4) to affine_map<(d0) -> (d0 + 512)>(%arg4) {
            affine.for %arg8 = affine_map<(d0) -> (d0)>(%arg5) to affine_map<(d0) -> (d0 + 8)>(%arg5) {
              %8 = affine.load %alloc_5[-%arg3 + %arg6, -%arg5 + %arg8] : memref<232x8xf64>
              %9 = affine.load %alloc_6[-%arg5 + %arg8, -%arg4 + %arg7] : memref<8x512xf64>
              %10 = affine.load %alloc[-%arg3 + %arg6, -%arg4 + %arg7] : memref<232x512xf64>
              %11 = arith.mulf %8, %9 : f64
              %12 = arith.addf %10, %11 : f64
              affine.store %12, %alloc[-%arg3 + %arg6, -%arg4 + %arg7] : memref<232x512xf64>
            }
          }
        }
        memref.dealloc %alloc_6 : memref<8x512xf64>
        memref.dealloc %alloc_5 : memref<232x8xf64>
      }
      %2 = affine.apply affine_map<(d0, d1) -> (d0)>(%arg3, %arg4)
      %3 = affine.apply affine_map<(d0, d1) -> (d1)>(%arg3, %arg4)
      affine.for %arg5 = affine_map<(d0) -> (d0)>(%arg3) to affine_map<(d0) -> (d0 + 232)>(%arg3) {
        affine.for %arg6 = affine_map<(d0) -> (d0)>(%arg4) to affine_map<(d0) -> (d0 + 512)>(%arg4) {
          %4 = affine.load %alloc[%arg5 - %arg3, %arg6 - %arg4] : memref<232x512xf64>
          affine.store %4, %arg2[%arg5, %arg6] : memref<2088x2048xf64>
        }
      }
      memref.dealloc %alloc : memref<232x512xf64>
    }
  }
  return
}
```

C.3 The -hlo-matmul-to-loops pass

The `-hlo-matmul-to-loops` pass transforms the `matmul` operation of the `hlo` dialect into a 3-d loop nest. This is demonstrated in listing C.5, which is the result of applying the pass to the input file in listing C.4. Listing C.6 presents the excerpt from listing C.5 corresponding to `hlo.matmul`.

Listing C.4: The demonstration source file `input/hlo_matmul_test.mlir`.

```
func.func @main() {
    %A = memref.alloc() : memref<2088x2048xf64>
    %B = memref.alloc() : memref<2048x2048xf64>
    %C = memref.alloc() : memref<2088x2048xf64>
    %cf1 = llvm.mlir.constant(1.00000e+00 : f64) : f64

    linalg.fill ins(%cf1 : f64) outs(%A : memref<2088x2048xf64>)
    linalg.fill ins(%cf1 : f64) outs(%B : memref<2048x2048xf64>)
    linalg.fill ins(%cf1 : f64) outs(%C : memref<2088x2048xf64>)

    hlo.matmul ins(%A : memref<2088x2048xf64>, %B : memref<2048x2048xf64>, %C :
    memref<2088x2048xf64>)

    memref.dealloc %A : memref<2088x2048xf64>
    memref.dealloc %B : memref<2048x2048xf64>
    memref.dealloc %C : memref<2088x2048xf64>

    return
}
```

Listing C.5: Output from running `mlir-opt -hlo-matmul-to-loops input/hlo_matmul_test.mlir`.

```
module {
  func.func @main() {
    %0 = llvm.mlir.constant(1.000000e+00 : f64) : f64
    %alloc = memref.alloc() : memref<2088x2048xf64>
    %alloc_0 = memref.alloc() : memref<2048x2048xf64>
    %alloc_1 = memref.alloc() : memref<2088x2048xf64>
    linalg.fill ins(%0 : f64) outs(%alloc : memref<2088x2048xf64>)
    linalg.fill ins(%0 : f64) outs(%alloc_0 : memref<2048x2048xf64>)
    linalg.fill ins(%0 : f64) outs(%alloc_1 : memref<2088x2048xf64>)
    affine.for %arg0 = 0 to 2088 {
      affine.for %arg1 = 0 to 2048 {
        affine.for %arg2 = 0 to 2048 {
          %1 = affine.load %alloc[symbol(%arg0), symbol(%arg2)] : memref<2088
    x2048xf64>
          %2 = affine.load %alloc_0[symbol(%arg2), symbol(%arg1)] : memref<2048
    x2048xf64>
          %3 = affine.load %alloc_1[symbol(%arg0), symbol(%arg1)] : memref<2088
    x2048xf64>
          %4 = arith.mulf %1, %2 fastmath<fast> : f64
          %5 = arith.addf %3, %4 fastmath<fast> : f64
          affine.store %5, %alloc_1[symbol(%arg0), symbol(%arg1)] : memref<2088
    x2048xf64>
        }
      }
    }
    memref.dealloc %alloc : memref<2088x2048xf64>
    memref.dealloc %alloc_0 : memref<2048x2048xf64>
    memref.dealloc %alloc_1 : memref<2088x2048xf64>
    return
  }
}
```

Listing C.6: Excerpt from listing C.5 which corresponds to `hlo.matmul`.

```
affine.for %arg0 = 0 to 2088 {
  affine.for %arg1 = 0 to 2048 {
    affine.for %arg2 = 0 to 2048 {
      %1 = affine.load %alloc[symbol(%arg0), symbol(%arg2)] : memref<2088x2048xf64>
      %2 = affine.load %alloc_0[symbol(%arg2), symbol(%arg1)] : memref<2048
    x2048xf64>
      %3 = affine.load %alloc_1[symbol(%arg0), symbol(%arg1)] : memref<2088
    x2048xf64>
      %4 = arith.mulf %1, %2 fastmath<fast> : f64
      %5 = arith.addf %3, %4 fastmath<fast> : f64
      affine.store %5, %alloc_1[symbol(%arg0), symbol(%arg1)] : memref<2088
    x2048xf64>
    }
  }
}
```

The loop nest is optimized by specifying the options within `-hlo-matmul-to-loops`.

The first option to demonstrate is `tile` which enables loop tiling, as shown in listing C.8. Listing C.7 presents the initial implementation of the loop tiling scheme, where the induction variables and loop bounds have not been adjusted according to our specifications.

Listing C.7: Output produced from the initial loop tiling implementation.

```
# map = affine_map <(d0) -> (d0)>
# map1 = affine_map <(d0) -> (2088 , d0 + 165)>
# map2 = affine_map <(d0) -> (2048 , d0 + 720)>
# map3 = affine_map <(d0) -> (d0 + 8)>
# map4 = affine_map <(d0 , d1) -> (d0 , d1)>
# map5 = affine_map <(d0 , d1) -> (2088 , d0 + 165 , d1 + 3)>
module {
  func.func @main() {
    %0 = llvm.mlir.constant (1.000000e+00 : f64 ) : f64
    %alloc = memref.alloc () : memref <2088x2048xf64>
    %alloc_0 = memref.alloc () : memref <2048x2048xf64>
    %alloc_1 = memref.alloc () : memref <2088x2048xf64>
    linalg.fill ins (%0 : f64) outs (% alloc : memref <2088 x2048xf64>)
    linalg.fill ins (%0 : f64) outs (% alloc_0 : memref <2048 x2048xf64>)
    linalg.fill ins (%0 : f64) outs (% alloc_1 : memref <2088 x2048xf64>)
    affine.for %arg0 = 0 to 2048 step 720 {
      affine.for %arg1 = 0 to 2088 step 165 {
        affine.for %arg2 = 0 to 2048 step 8 {
          affine.for %arg3 = #map (%arg1) to min #map1 (%arg1) step 3 {
            affine.for %arg4 = #map (%arg0) to min #map2 (%arg0) {
              affine.for %arg5 = #map (%arg2) to #map3 (% arg2 ) {
                affine.for %arg6 = max #map4 (%arg1 , %arg3) to min #map5 (%arg1 , %
    arg3) {
                  %1 = affine.load %alloc [symbol (%arg6), symbol (%arg4) ] :
    memref<2088x2048xf64>
                  %2 = affine.load %alloc_0[symbol (%arg4), symbol (%arg5) ] :
    memref<2048x2048xf64>
                  %3 = affine.load %alloc_1 [symbol (%arg6), symbol (%arg5) ] :
    memref<2088x2048xf64>
                  %4 = arith.mulf %1, %2 fastmath <fast> : f64
                  %5 = arith.addf %3, %4 fastmath <fast> : f64
                affine.store %5, %alloc_1 [symbol (% arg6), symbol (% arg5)] :
    memref<2088x2048xf64>
                }
              }
            }
          }
        }
      }
    }
    memref.dealloc %alloc : memref <2088 x2048xf64>
    memref.dealloc %alloc_0 : memref <2048 x2048xf64>
    memref.dealloc %alloc_1 : memref <2088 x2048xf64>
    return
  }
}
```

Listing C.8: Loop tiling enabled with $M_C = 165$, $K_C = 720$, $M_R = 3$ and $N_R = 8$.

```
#map = affine_map<(d0) -> (d0 * 55)>
#map1 = affine_map<(d0) -> (696, d0 * 55 + 55)>
#map2 = affine_map<(d0) -> (d0 * -720 + 2048, 720)>
module {
  func.func @main() {
    %0 = llvm.mlir.constant(1.000000e+00 : f64) : f64
    %alloc = memref.alloc() : memref<2088x2048xf64>
    %alloc_0 = memref.alloc() : memref<2048x2048xf64>
    %alloc_1 = memref.alloc() : memref<2088x2048xf64>
    linalg.fill ins(%0 : f64) outs(%alloc : memref<2088x2048xf64>)
    linalg.fill ins(%0 : f64) outs(%alloc_0 : memref<2048x2048xf64>)
    linalg.fill ins(%0 : f64) outs(%alloc_1 : memref<2088x2048xf64>)
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 13 {
        affine.for %arg2 = 0 to 256 {
          affine.for %arg3 = #map(%arg1) to min #map1(%arg1) {
            affine.for %arg4 = 0 to min #map2(%arg0) {
              affine.for %arg5 = 0 to 8 {
                affine.for %arg6 = 0 to 3 {
                  %1 = affine.load %alloc[%arg6 + %arg3 * 3, %arg4 + %arg0 * 720] :
     memref<2088x2048xf64>
                  %2 = affine.load %alloc_0[%arg4 + %arg0 * 720, %arg5 + %arg2 * 8]
     : memref<2048x2048xf64>
                  %3 = affine.load %alloc_1[%arg6 + %arg3 * 3, %arg5 + %arg2 * 8] :
     memref<2088x2048xf64>
                  %4 = arith.mulf %1, %2 fastmath<fast> : f64
                  %5 = arith.addf %3, %4 fastmath<fast> : f64
                  affine.store %5, %alloc_1[%arg6 + %arg3 * 3, %arg5 + %arg2 * 8] :
     memref<2088x2048xf64>
                }
              }
            }
          }
        }
      }
    }
    memref.dealloc %alloc : memref<2088x2048xf64>
    memref.dealloc %alloc_0 : memref<2048x2048xf64>
    memref.dealloc %alloc_1 : memref<2088x2048xf64>
    return
  }
}
```

Option `copy` enables explicit copying, as shown in listing C.9.

Listing C.9: Loop tiling and explicit copying enabled.

```
#map = affine_map<(d0) -> (d0 * 165)>
#map1 = affine_map<(d0) -> (2088, d0 * 165 + 165)>
#map2 = affine_map<(d0) -> (d0 * 720)>
#map3 = affine_map<(d0) -> (2048, d0 * 720 + 720)>
#map4 = affine_map<(d0) -> (d0 * 8)>
#map5 = affine_map<(d0) -> (d0 * 8 + 8)>
#map6 = affine_map<(d0) -> (d0 * 55)>
#map7 = affine_map<(d0) -> (696, d0 * 55 + 55)>
#map8 = affine_map<(d0) -> (d0 * -720 + 2048, 720)>
module {
  func.func @main() {
    %0 = llvm.mlir.constant(1.000000e+00 : f64) : f64
    %alloc = memref.alloc() : memref<2088x2048xf64>
    %alloc_0 = memref.alloc() : memref<2048x2048xf64>
    %alloc_1 = memref.alloc() : memref<2088x2048xf64>
    linalg.fill ins(%0 : f64) outs(%alloc : memref<2088x2048xf64>)
    linalg.fill ins(%0 : f64) outs(%alloc_0 : memref<2048x2048xf64>)
    linalg.fill ins(%0 : f64) outs(%alloc_1 : memref<2088x2048xf64>)
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 13 {
        %alloc_2 = memref.alloc() {alignment = 32 : i64} : memref<165x720xf64>
        affine.for %arg2 = #map(%arg1) to min #map1(%arg1) {
          affine.for %arg3 = #map2(%arg0) to min #map3(%arg0) {
            %1 = affine.load %alloc[%arg2, %arg3] : memref<2088x2048xf64>
            affine.store %1, %alloc_2[%arg2 - %arg1 * 165, %arg3 - %arg0 * 720] :
    memref<165x720xf64>
          }
        }
        affine.for %arg2 = 0 to 256 {
          %alloc_3 = memref.alloc() {alignment = 32 : i64} : memref<720x8xf64>
          affine.for %arg3 = #map2(%arg0) to min #map3(%arg0) {
            affine.for %arg4 = #map4(%arg2) to #map5(%arg2) {
              %1 = affine.load %alloc_0[%arg3, %arg4] : memref<2048x2048xf64>
              affine.store %1, %alloc_3[%arg3 - %arg0 * 720, %arg4 - %arg2 * 8] :
    memref<720x8xf64>
            }
          }
          affine.for %arg3 = #map6(%arg1) to min #map7(%arg1) {
            affine.for %arg4 = 0 to min #map8(%arg0) {
              affine.for %arg5 = 0 to 8 {
                affine.for %arg6 = 0 to 3 {
                  %1 = affine.load %alloc_2[%arg1 * -165 + %arg6 + %arg3 * 3, %arg4] : memref<165x720xf64>
                  %2 = affine.load %alloc_3[%arg4, %arg5] : memref<720x8xf64>
                  %3 = affine.load %alloc_1[%arg6 + %arg3 * 3, %arg5 + %arg2 * 8] : memref<2088x2048xf64>
                  %4 = arith.mulf %1, %2 fastmath<fast> : f64
                  %5 = arith.addf %3, %4 fastmath<fast> : f64
                  affine.store %5, %alloc_1[%arg6 + %arg3 * 3, %arg5 + %arg2 * 8] : memref<2088x2048xf64>
                }
              }
            }
          }
          memref.dealloc %alloc_3 : memref<720x8xf64>
        }
        memref.dealloc %alloc_2 : memref<165x720xf64>
      }
    }
    memref.dealloc %alloc : memref<2088x2048xf64>
    memref.dealloc %alloc_0 : memref<2048x2048xf64>
    memref.dealloc %alloc_1 : memref<2088x2048xf64>
    return
  }
}
```

Listing C.10 demonstrates loop unrolling with scalar replacement, enabled through `unroll` and `scalar-replace`. Explicit copying has been disabled, N_R has been decreased and K_U is set to 1 to minimize the size of the output.

Listing C.10: Loop tiling, loop unrolling and scalar replacement enabled. $N_R = 2$ and $K_U = 1$ for readability.

```
#map = affine_map<(d0) -> (d0 * 55)>
#map1 = affine_map<(d0) -> (696, d0 * 55 + 55)>
#map2 = affine_map<(d0) -> (d0 * -720 + 2048, 720)>
module {
  func.func @main() {
    %0 = llvm.mlir.constant(1.000000e+00 : f64) : f64
    %alloc = memref.alloc() : memref<2088x2048xf64>
    %alloc_0 = memref.alloc() : memref<2048x2048xf64>
    %alloc_1 = memref.alloc() : memref<2088x2048xf64>
    linalg.fill ins(%0 : f64) outs(%alloc : memref<2088x2048xf64>)
    linalg.fill ins(%0 : f64) outs(%alloc_0 : memref<2048x2048xf64>)
    linalg.fill ins(%0 : f64) outs(%alloc_1 : memref<2088x2048xf64>)
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 13 {
        affine.for %arg2 = 0 to 1024 {
          affine.for %arg3 = #map(%arg1) to min #map1(%arg1) {
            affine.for %arg4 = 0 to min #map2(%arg0) {
              %1 = affine.load %alloc[%arg3 * 3, %arg4 + %arg0 * 720] : memref<2088x2048xf64>
              %2 = affine.load %alloc_0[%arg4 + %arg0 * 720, %arg2 * 2] : memref<2048x2048xf64>
              %3 = affine.load %alloc_1[%arg3 * 3, %arg2 * 2] : memref<2088x2048xf64>
              %4 = arith.mulf %1, %2 fastmath<fast> : f64
              %5 = arith.addf %3, %4 fastmath<fast> : f64
              affine.store %5, %alloc_1[%arg3 * 3, %arg2 * 2] : memref<2088x2048xf64>
              %6 = affine.load %alloc[%arg3 * 3 + 1, %arg4 + %arg0 * 720] : memref<2088x2048xf64>
              %7 = affine.load %alloc_1[%arg3 * 3 + 1, %arg2 * 2] : memref<2088x2048xf64>
              %8 = arith.mulf %6, %2 fastmath<fast> : f64
              %9 = arith.addf %7, %8 fastmath<fast> : f64
              affine.store %9, %alloc_1[%arg3 * 3 + 1, %arg2 * 2] : memref<2088x2048xf64>
              %10 = affine.load %alloc[%arg3 * 3 + 2, %arg4 + %arg0 * 720] : memref<2088x2048xf64>
              %11 = affine.load %alloc_1[%arg3 * 3 + 2, %arg2 * 2] : memref<2088x2048xf64>
              %12 = arith.mulf %10, %2 fastmath<fast> : f64
              %13 = arith.addf %11, %12 fastmath<fast> : f64
              affine.store %13, %alloc_1[%arg3 * 3 + 2, %arg2 * 2] : memref<2088x2048xf64>
              %14 = affine.load %alloc_0[%arg4 + %arg0 * 720, %arg2 * 2 + 1] : memref<2048x2048xf64>
              %15 = affine.load %alloc_1[%arg3 * 3, %arg2 * 2 + 1] : memref<2088x2048xf64>
              %16 = arith.mulf %1, %14 fastmath<fast> : f64
              %17 = arith.addf %15, %16 fastmath<fast> : f64
              affine.store %17, %alloc_1[%arg3 * 3, %arg2 * 2 + 1] : memref<2088x2048xf64>
              %18 = affine.load %alloc_1[%arg3 * 3 + 1, %arg2 * 2 + 1] : memref<2088x2048xf64>
              %19 = arith.mulf %6, %14 fastmath<fast> : f64
              %20 = arith.addf %18, %19 fastmath<fast> : f64
              affine.store %20, %alloc_1[%arg3 * 3 + 1, %arg2 * 2 + 1] : memref<2088x2048xf64>
              %21 = affine.load %alloc_1[%arg3 * 3 + 2, %arg2 * 2 + 1] : memref<2088x2048xf64>
              %22 = arith.mulf %10, %14 fastmath<fast> : f64
              %23 = arith.addf %21, %22 fastmath<fast> : f64
              affine.store %23, %alloc_1[%arg3 * 3 + 2, %arg2 * 2 + 1] : memref<2088x2048xf64>
            }
          }
        }
      }
    }
    memref.dealloc %alloc : memref<2088x2048xf64>
    memref.dealloc %alloc_0 : memref<2048x2048xf64>
    memref.dealloc %alloc_1 : memref<2088x2048xf64>
    return
  }
}
```

Listing C.11 corresponds to listing C.10 with vectorization enabled.

Listing C.11: Loop tiling, loop unrolling, scalar replacement and vectorization enabled. $N_R = 2$ and $K_U = 1$ for readability.

```
#map = affine_map<(d0) -> (d0 * 55)>
#map1 = affine_map<(d0) -> (696, d0 * 55 + 55)>
#map2 = affine_map<(d0) -> (d0 * -720 + 2048, 720)>
module {
  func.func @main() {
    %0 = llvm.mlir.constant(1.000000e+00 : f64) : f64
    %alloc = memref.alloc() : memref<2088x2048xf64>
    %alloc_0 = memref.alloc() : memref<2048x2048xf64>
    %alloc_1 = memref.alloc() : memref<2088x2048xf64>
    linalg.fill ins(%0 : f64) outs(%alloc : memref<2088x2048xf64>)
    linalg.fill ins(%0 : f64) outs(%alloc_0 : memref<2048x2048xf64>)
    linalg.fill ins(%0 : f64) outs(%alloc_1 : memref<2088x2048xf64>)
    %1 = memref.shape_cast %alloc_0 : memref<2048x2048xf64> to memref<2048x1024xvector<2xf64>>
    %2 = memref.shape_cast %alloc_1 : memref<2088x2048xf64> to memref<2088x1024xvector<2xf64>>
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 13 {
        affine.for %arg2 = 0 to 1024 {
          affine.for %arg3 = #map(%arg1) to min #map1(%arg1) {
            affine.for %arg4 = 0 to min #map2(%arg0) {
              %3 = affine.load %alloc[%arg3 * 3, %arg4 + %arg0 * 720] : memref<2088x2048xf64>
              %4 = vector.splat %3 : vector<2xf64>
              %5 = affine.load %1[%arg4 + %arg0 * 720, %arg2] : memref<2048x1024xvector<2xf64>>
              %6 = affine.load %2[%arg3 * 3, %arg2] : memref<2088x1024xvector<2xf64>>
              %7 = vector.fma %4, %5, %6 : vector<2xf64>
              affine.store %7, %2[%arg3 * 3, %arg2] : memref<2088x1024xvector<2xf64>>
              %8 = affine.load %alloc[%arg3 * 3 + 1, %arg4 + %arg0 * 720] : memref<2088x2048xf64>
              %9 = vector.splat %8 : vector<2xf64>
              %10 = affine.load %1[%arg4 + %arg0 * 720, %arg2] : memref<2048x1024xvector<2xf64>>
              %11 = affine.load %2[%arg3 * 3 + 1, %arg2] : memref<2088x1024xvector<2xf64>>
              %12 = vector.fma %9, %10, %11 : vector<2xf64>
              affine.store %12, %2[%arg3 * 3 + 1, %arg2] : memref<2088x1024xvector<2xf64>>
              %13 = affine.load %alloc[%arg3 * 3 + 2, %arg4 + %arg0 * 720] : memref<2088x2048xf64>
              %14 = vector.splat %13 : vector<2xf64>
              %15 = affine.load %1[%arg4 + %arg0 * 720, %arg2] : memref<2048x1024xvector<2xf64>>
              %16 = affine.load %2[%arg3 * 3 + 2, %arg2] : memref<2088x1024xvector<2xf64>>
              %17 = vector.fma %14, %15, %16 : vector<2xf64>
              affine.store %17, %2[%arg3 * 3 + 2, %arg2] : memref<2088x1024xvector<2xf64>>
            }
          }
        }
      }
    }
    memref.dealloc %alloc : memref<2088x2048xf64>
    memref.dealloc %alloc_0 : memref<2048x2048xf64>
    memref.dealloc %alloc_1 : memref<2088x2048xf64>
    return
  }
}
```

DEPARTMENT OF COMPUTER SCIENCE | LUNDS TEKNISKA HÖGSKOLA | PRESENTED 2024-01-09

MASTER'S THESIS MLIR-based Code Generation for High-Performance Machine Learning on AArch64
STUDENT Johanna Gustafson
SUPERVISORS Jonas Skeppstedt (LTH), Fredrik Knutsson (Arm), Per Åstrand (Arm)
EXAMINER Flavius Gruian (LTH)

Using MLIR to conquer machine learning

POPULAR SCIENCE SUMMARY **Johanna Gustafson**

In the quest of high execution speeds, current machine learning systems tend to rely on high-performance compute libraries optimized for a narrow range of hardware devices. Thus, we propose a modular approach in the design of machine learning systems, utilizing the reusable and extensible MLIR compiler framework.

Imagine you are in middle school, given the assignment of constructing a durable miniature bridge. The more weight it can bear, the better. Your teacher explains that the best bridges tend to be based upon triangular units and mentions the *Howe truss* and *Warren truss* bridges. With no further explanation, they walk out of the room. Suddenly, you feel confused; what is Howe or Warren truss? Not only do you have to come up with a design through guesswork, you also have to collect material on your own.

Our master's thesis investigates this issue, although replacing bridges with machine learning systems and durability with performance. More specifically, the problem pertains to adapting advancements within machine learning technology to the ever-expanding plethora of hardware. We believe MLIR (Multi-Level Intermediate Representation) — a novel approach to building reusable and extensible compiler infrastructure — is a solution to this issue. MLIR can be likened to the teacher providing you and your classmates with popsicle sticks, glue and a suggested step-by-step guide on how to go about the task, although giving you the freedom to modify and expand as you wish. This simple consideration eliminates the effort by you and your classmates to each surpass the initial threshold of the assignment.

Given the width of machine learning topics, we focused on one fundamental computation — multiplication and addition of matrices — and demonstrated the implementation of an optimized MLIR operation, as presented in figure 1. We believe the results of our research is indicative of the potential benefits of incorporating MLIR into machine learning systems, in constituting retargetability while retaining high performance.

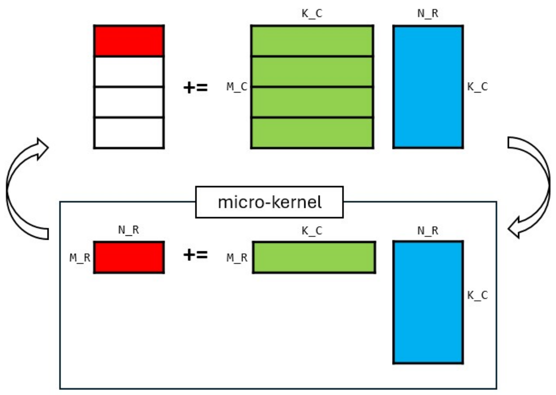


Figure 1: The implemented operation is essentially a nest of loops around a so-called *micro-kernel*; a small sequence of computation tasks. This image illustrates the two innermost loops, where partitions of the three original matrices are multiplied and added.