Efficient Memory Hierarchy Utilization for Matrix Multiplication and Convolution on CPUs

by

Ivan Korostelev

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Matrix multiplication is a key operation both in high-performance computing and in deep-learning applications. Generic building blocks have been introduced to abstract a matrix-multiplication operation and to enable the generation of efficient code for multiple architectures. This thesis presents three solutions to create software stacks that connect user-level code with the use of these building blocks. The first one uses pattern recognition at the intermediate representation level to discover matrix routines and replace them with efficient hand-crafted implementations that exist in numerical libraries. The second solution is a compiler-only code-generation path for matrix multiplication that targets multiple platforms. The third one is a novel convolution algorithm that replaces the traditional image-to-column data transformation with a custom cache-utilization strategy. Common to all three solutions are the use of generic building blocks that are target agnostic and the focus on making efficient use of the memory hierarchy of each machine.

The first solution, KernelFaRer, is a compiler optimization pass that searches the source code for linear algebra routines, such as matrix multiplication and rank-2 matrix update, and replaces these occurrences with efficient library implementations. KernelFaRer operates on the LLVM's intermediate representation language which makes the tool source-language agnostic and granular enough to recognize complex data dependency patterns. In comparison with other pattern matching approaches, KernelFaRer is more robust and carries less compilation time overhead.

The second solution brings the ideas for efficient matrix multiplication from high-performance libraries into a production LLVM compiler. The evaluation study shows that this approach delivers performance matching that of the high-performance libraries. This approach allows efficient linear algebra algorithms before high-performance libraries officially release them, as was the case for the IBM POWER10™ architecture.

The third solution, YaConv, is a novel convolution algorithm that repurposes the building blocks for matrix multiplication from a popular high-performance library. The custom cache utilization strategy reduces the number of cache accesses by 5× and achieves mean speedup of 15% over the standard im2col-based convolution. Moreover, by integrating placement of elements into cache with their use by vector instructions, the new algorithm requires only a constant buffer of the size of cache, as compared to huge memory consumption of the traditional convolution algorithms.

Preface

Chapter 3 of this thesis was published as J. P. L. De Carvalho, B. Kuzma, I. Korostelev, J. N. Amaral, C. Barton, J. Moreira, and G. Araujo, "KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls", ACM Transactions on Architecture and Code Optimization, Volume 18, Issue 3, September 2021. My role on the project was to reduce false negatives of the pattern matching tool for GEMM, extend it to SYR2K and find the cause for false negatives in matcher results from LLVM Polly. The core of the matcher design was developed by J. P. L. De Carvalho and B. Kuzma. J .N. Amaral and G.Araujo were the supervisory authors and contributed to guiding the experimental evaluation and editing the resulting manuscript. C. Barton and J. Moreira were the IBM collaborators and participated in technical discussions.

Chapter 4 of this thesis has been submitted for publication as B. Kuzma, I. Korostelev, J. P. L. De Carvalho, J. Moreira, C. Barton, G. Araujo, J. N. Amaral, "Fast Matrix Multiplication via Compiler-only Layered Data Reorganization and Intrinsic Lowering". I was responsible for developing the matrix multiplication algorithm, together with B. Kuzma, who developed the target-specific part of the code. I was also responsible for literature review and helped J. P. L. De Carvalho perform the experimental evaluation and report the results. C. Barton and J. Moreira were the IBM collaborators and participated in technical discussions. G.Araujo and J .N. Amaral were the supervisory authors and contributed to guiding the experimental evaluation and editing the resulting manuscript.

Chapter 5 of this thesis has been submitted for publication as I. Korostelev, J. P. L. De Carvalho, J. Moreira, J. N. Amaral, "YaConv: Convolution with Low Cache Footprint". I was responsible for designing and developing the algorithm,

performing the experimental evaluation, literature review and drafting the manuscript. J. P. L. De Carvalho contributed to algorithm design and the resulting manuscript. J. Moreira provided knowledge on the architectural design of IBM machines and helped with running the experiments. J .N. Amaral was the supervisory author and contributed to guiding the experimental evaluation and editing the resulting manuscript.

To my dad, for the discipline.

To my mom, for the unconditional love.

The only way to do great work is to love what you do.

– Steve Jobs

$419 > 420$

Acknowledgements

I thank my supervisor, Dr. J Nelson Amaral, for teaching me how to speak. I appreciate the many times he has pushed me to do things the right way and not cut the corners.

I thank my friend João for the insightful conversations on computers, a gentle introduction to LLVM and many cool tips and tricks.

I thank Karolína for the support she offered during the slow-progressing periods of my thesis.

I thank my friends Cody, Batyr, Logan, Wilson, Tomas and João who have always offered a great company to relax from work.

I thank Ford Motor Company for making a great Lincoln Town Car which let me explore the Canadian Rockies and reset my mind so many times.

Thanks to everyone who joined me outdoors!

I thank my colleague Braedy Kuzma for his help with algorithm development.

I thank the IBM collaborators, José Moreira and Kit Barton, for providing their knowledge of the IBM hardware and infrastructure.

I thank Dr. Guido Araujo and Victor Ferrari for the conversations that shaped my work on convolution.

This research has been funded in part by the IBM Center for Advanced Studies (CAS) Canada and Graduate Research Assistantship Fellowship (GRAF).

Contents

List of Tables

List of Figures

List of Acronyms

AI Artificial Intelligence

AMX Advanced Matrix Extensions

BLAS Basic Linear Algebra Subprograms

CNN Convoutional Neural Network

CPU Central Processing Unit

GEMM General Matrix Multiplication

HPC High-Performance Computing

IR Intermediate Representation

LLVM Low-Level Virtual Machine

MLIR Multi-Level Intermediate Representation

MMA Matrix Math Assist

SIMD Single Instruction, Multiple Data

TLB Translation Lookaside Buffer

VSX Vector-Scalar Extension

Chapter 1

Introduction

The Artificial Intelligence (AI) resurgence of the last decade has increased demand for both server-grade and mobile computing power. Convoutional Neural Network (CNN) is the central algorithm for pattern recognition tasks, such as image classification, speech processing and language translation. More than 95% of the CNN training and inference time is spent in matrix multiplication of the fully-connected and convolutional layers [55]. Numerous engineering and data processing algorithms, such as matrix decomposition, Principal Component Analysis (PCA) and k-Means continue to heavily rely on matrix multiplication. To meet the surging demand, major chip vendors integrate new hardware solutions for matrix operations. IBM® POWER10 features MMA units which are intended to accelerate matrix operations on data of any type stored in vector registers [84]. Intel® introduced Advanced Matrix Extensions (AMX) designed to work on 8- or 16-bit matrix data stored in two-dimensional registers, called tiles [61]. The differences in matrix accelerator unit implementations in the hardware trickle down the software stack. With more development in specialized hardware for AI, an explosion in the number of code generation paths will require new approaches.

For several decades, efficient code for the critical computational primitives has been provided by specialized numerical libraries. With the explosion in the variety of hardware architectures available on the market, the task of writing up-to-date assembly code for each platform becomes unsurmountable. This has spurred advances in the compiler technologies with Low-Level Virtual

Machine (LLVM) being the most prominent compiler infrastructure for domain-specific applications. Polyhedral loop analysis, network graph optimizations and heterogenous hardware code generation are all now avalaible as part of LLVM. The flexibility of the project allows to add new architecture targets promptly and efficiently. Multi-Level Intermediate Representation (MLIR), a project based on the LLVM infrastructure, has already integrated Intel's recently-added Advanced Matrix Extensions and compiler developers have integrated these changes in the exisiting numerical primitives [32].

Chapter 2

Background

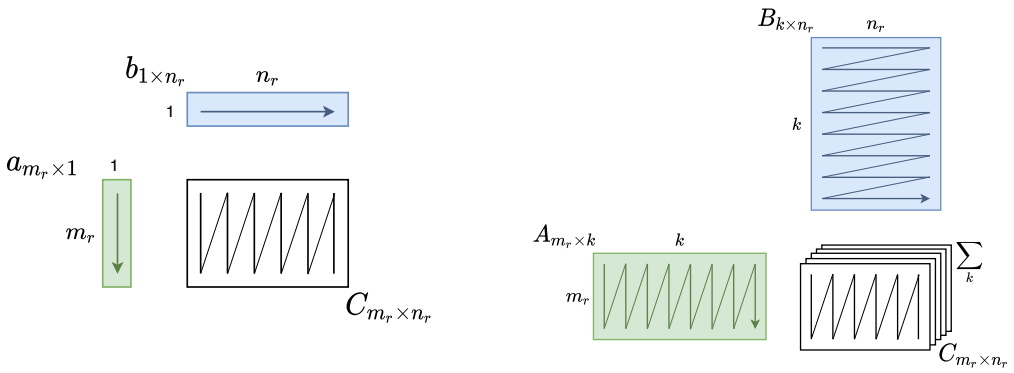
General Matrix Multiplication (GEMM) is a standard routine from Basic Linear Algebra Subprograms (BLAS) that has been optimized for over 40 years and has close to peak performance implementations in the open-source libraries such as BLIS and OpenBLAS [70], [94]. In mathematical notation, GEMM is expressed by the formula:

$$C = \beta \cdot C + \alpha \cdot A * B \tag{2.1}$$

where α and β are scalars, $A_{M \times K}$ and $B_{K \times N}$ are the input matrices, and $C_{M \times N}$ is the output matrix. Most high-performance implementations of GEMM rely on the seminal work of Goto and Geijn [35]. Peak Central Processing Unit (CPU) performance for GEMM is achieved by a loop nest that optimizes data cache and Translation Lookaside Buffer (TLB) locality and leverages an efficient GEMM microkernel. Throughout the thesis, I refer to this algorithm as the conventional GEMM algorithm.

2.1 Outer Product

Matrix multiplication is often introduced as the computation of multiple inner products, as defined by the sum $C[i][j] = \sum_{k=1}^{K} A[i][k] \cdot B[k][j]$. Implementations of GEMM directly using this inner product form suffer from poor reuse of loaded register values. Instead, the GEMM microkernel in BLAS libraries is implemented as multiple outer product computations.



(a) One outer product update

(b) GEMM microkernel is a rank-k update

Figure 2.1: GEMM as outer product

Figure 2.1a shows one outer product update (rank-1 update) that computes partial result $C_{m_r\times n_r} \mathrel{+}= a_{m_r\times 1} * b_{1\times n_r}$. In each update, elements of the vector $a_{m_r\times 1}$ and elements of the vector $b_{1\times n_r}$ are loaded into vector registers. Either elements of a or b are broadcast in registers to produce an operand tile of size $m_r \times 1$ or $1 \times n_r$. The values in registers are multiplied and accumulated to $C_{m_r\times n_r}$. To compute the full GEMM, this step is repeated for each column of $A_{m_r\times k}$ and each row of $B_{k\times n_r}$, as shown in Figure 2.1b.

Unrolling the loop along k dimension and prefetching the next elements are commonly implemented to achieve better performance [35], [70], [94]. After k-loop unrolling, several columns of A and rows of B are used for each update to maximize vector-register utilization. Sizes m_r and n_r control the amount of register reuse by the outer product update [35]. These two parameters depend on the architecture and define the minimum GEMM size the microkernel will compute at peak performance. In order to utilize the microkernel for the full GEMM, a cache-aware strategy must tile the arrays into cache-sized buffers. Moreover, the elements should be placed in the buffers in the order in which they will be accessed by the outer-product updates, as shown in Figure 2.1b.

2.2 Accelerators With Outer and Inner Product

Matrix multiplication and other linear algebra operations can be expressed as a series of rank-k update operations. These operations compute the product of an $m \times k$ matrix by another $k \times n$ matrix, accumulating the result into an $m \times n$ matrix. When $k = 1$, the operation reduces to an outer product of two vectors, of m and n elements, respectively as illustrated in Figure 4.1. A rank-k update can itself be decomposed to a sequence of k outer products.

2.2.1 Matrix-Multiply Assist in POWER10

The MMA instructions were introduced in PowerISA 3.1 [50] as an extension of the Vector-Scalar Extension (VSX) facility. IBM's POWER10 processor is the first to implement these new instructions and functional units that perform two-dimensional matrix operations. MMA relies on 512-bit accumulator (ACC) registers to represent matrices, which can be manipulated by BLAS-like rank-k operations that consume vector registers as inputs. Each accumulator register is associated with four of the architecture's 128-bit vector-scalar registers (VSRs). While an accumulator is being used for MMA instructions, the associated VSRs are blocked from use. Up to eight of these accumulators can be use simultaneously, leaving 32 of 64 VSRs available for use as vector registers.

Outer-products have high-computational density since they are two-dimensional operations that compute mn element-wise operations from $m + n$ input values. Therefore, they are the standard building block of high-performance linear algebra frameworks such as OpenBLAS and Eigen. In processors with one-dimensional vector instructions, the outer products are emulated using a combination of broadcasting and element-wise multiply-add instructions. MMA bypasses this emulation step by directly supporting outer product.

In the MMA rank-k update instructions, the updated matrix is stored in an ACC, while the operand vectors (or matrices) are provided through VSRs. The result matrix is either a 4×4 matrix of 32-bit elements (floating-point or

Table 2.1: MMA instruction summary.

Input type	Computation size $m \times k \cdot k \times n$	Result shape and type
4-bit integer (`i4`)	$4 \times 8 \cdot 8 \times 4$	4×4 `i32`
8-bit integer (`i8`)	$4 \times 4 \cdot 4 \times 4$	
16-bit integer (`i16`)	$4 \times 2 \cdot 2 \times 4$	
brain-float (`bf16`)	$4 \times 2 \cdot 2 \times 4$	4×4 `f32`
IEEE half-precision (`f16`)	$4 \times 2 \cdot 2 \times 4$	
IEEE single-precision (`f32`)	$4 \times 1 \cdot 1 \times 4$	
IEEE double-precision (`f64`)	$4 \times 1 \cdot 1 \times 2$	4×2 `f64`

integer) or a 4×2 matrix of 64-bit floating point elements. The k is a function of the input data type, which can vary from 4-bit integers ($k = 8$) to 32- or 64-bit floating-point numbers ($k = 1$). For all input data types of 32-bit or less, the multiplying operands are represented by one VSR each. For 64-bit inputs, one operand is represented by a pair of VSRs ($4 \times$ 64-bit elements) and the other by a single VSR ($2 \times$ 64-bit elements). A summary of the MMA rank-k update instructions is shown in Table 2.1.

MMA can operate with several data types that have different sizes. Moreover, a single MMA instruction can accumulate multiple outer products depending on the size of the data elements. For instance, for a 32-bit data type, each 128-bit VSR contains four elements and the MMA instruction computes and accumulates a single outer product into the 512-bit accumulator that contains 4×4 matrix elements [7]. Following BLAS terminology, such an instruction is called a rank 1 update [29]. However, for a 16-bit data type, each VSR contains eight elements and the MMA instruction computes and accumulates two outer products into the 4×4-element accumulator, thus computing a rank 2 update. For 8-bit data types the instruction computes rank 4 updates and for 4-bit data types it computes rank 8 updates. Table 2.1 shows the types supported by MMA. The computation size indicates the size of each operand and the rank of the update computed by an MMA instruction. For all types with up to 32 bits the result in the accumulator is 16 32-bit values organized in a 4×4 grid. For `f64` the accumulator contains 4×2 elements of the matrix and performs a rank 1 update.

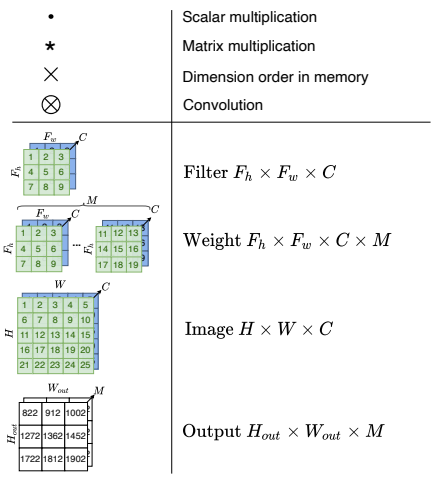
2.2.2 Accelerators With Inner Product

IBM's POWER10 MMA uses outer product as a base operation, which allows for reuse of vector registers. In contrast, the Single Instruction, Multiple Data (SIMD) processing unity of IBM's Power9™ VSX, and early versions, only feature inner-product operations. In the POWER10 MMA, after column elements of a matrix `A` and row elements of a matrix `B` are loaded into VSRs, each outer-product instruction computes a partial product of a piece of $\texttt{A} \times \texttt{B}$. Each VSR lane of an MMA accumulator holds accumulated partial products of different cells of the resulting matrix `C`. Performing the accumulation in place in MMA eliminates the need for any additional data movement. In contrast, VSX inner product multiplies rows of matrix `A` by columns of matrix `B` which were loaded into VSRs. Each lane of the resulting inner-product VSR has an element-wise product. The final value of a cell of `C` is computed by adding each VSR lane together, in what is called a horizontal reduction.

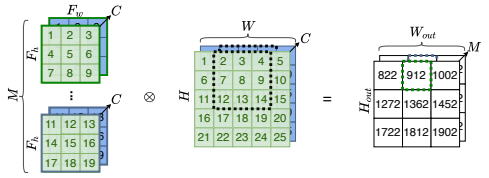
Horizontal reductions are more expensive than vertical reductions[1] because their latency is proportional to a SIMD vector length [92]. The across-lane computation required for a horizontal reduction defeats the goal of parallel computation on each SIMD lane in vectorization. To make a better usage of vector instructions, efficient linear-algebra libraries, such as OpenBLAS [93] and Eigen [38], emulate an outer-product computation by changing how the input elements are loaded into SIMD vector lanes. This emulation technique consists in broadcasting a single column element of matrix `A` over all SIMD lanes of a vector register to multiply it with a vector register with row elements of B. This strategy is more performant than direct inner-product matrix multiplication. However, it wastes vector register space and reduces the utilization of the level 1 cache because neighboring elements that are already in the cache are not readily loaded into registers.

Both Intel's AMX [53] and Arm's ME only support inner-product operations. AMX computes a block of the resulting matrix `C` through an inner-product

[1] Also known as element-wise reductions.



(a) Notation



(b) Naive convolution example

Figure 2.2: Convolution Notation and an Example of Naive Convolution

multiplication of 2D tiles of matrix `A` and matrix `B`. Arm's ME can emulate outer product using the broadcasting technique used in libraries.

2.3 Convolution Notation

Convolutional neural networks consist of layers, each of which has a fixed weight tensor. Each convolutional layer can be expressed by the formula:

$$W_{F_h \times F_w \times C \times M} \otimes I_{H \times W \times C} = O_{H_{out} \times W_{out} \times M} \tag{2.2}$$

where $I_{H \times W \times C}$ and $O_{H_{out} \times W_{out} \times M}$ are the input and output tensors and $W_{F_h \times F_w \times C \times M}$ is the weight tensor.

Figure 2.2a illustrates the tensor notation and the meaning of symbols $\cdot$, $*$, $\otimes$, and $\times$ used throughout this thesis. A variable input to each layer is the tensor $I_{H \times W \times C}$ that contains an image of height H, width W and the number of input channels C. Shown in Figure 2.2a, a weight tensor has shape $F_h \times F_w \times C \times M$ and consists of M filters. Each filter is a tensor of shape $[F_h \times F_w \times C]$, where C is the number of channels, F_h is the height of the filter, and F_w its width. The output tensor has dimensions $[H_{out} \times W_{out} \times M]$, where H_{out} and W_{out} are the output image height and width. Vertical and horizontal padding P_h, P_w with zero-elements are typically applied to enlarge the input image so that the output image is of the same size ($H_{out} = H, W_{out} = W$).

Chapter 3

KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls

General matrix-matrix multiplication (GEMM) and **symmetric rank-2k update (SYR2K)** are simple to implement naively but complex when designed to optimize the memory hierarchy [35]. GEMM in particular is idiomatic because it can be succinctly expressed and it exhibits a direct relation between implementation complexity and performance [74]. Most generic compiler loop transformations fail to exploit specific features of programming idioms such as GEMM [12], [37]. This chapter reaffirms that both sophisticated programmers and compilers still do not generate code with the performance of well-tuned libraries [34]. Thus, instead of optimizing an idiom, a compiler may simply replace it with a call to a library such as the **Basic Linear Algebra Subprograms (BLAS)** library [9]. These expertly crafted implementations efficiently exploit the memory hierarchy and can deliver high throughput on their target platform [22], [35], [93]. This chapter argues that idiom replacement must be both *robust* and *safe* in order to be an effective solution.

Despite progress in recent research [18], [34], existing approaches for idiom identification are brittle and fail to recognize minor variants of an idiom. Existing solutions require intimate knowledge of polyhedral analysis [18] or of a new domain-specific language that describes idioms [34]. They also have high compilation time costs even when the target idiom is absent from the code

(see Section 3.3.5).

Our solution is a new optimization pass for the LLVM framework [64]. This pass combines tree matching and idiom recognition [14], [68], [74], [75] with a data dependence analysis to deliver a robust idiom recognition and a safe replacement strategy. Previously, Carvalho et al. presented a work-in-progress report of this compiler pass [15]. This solution extends tooling already present in the LLVM framework and is fully integrated in this widely used compiler framework. Therefore it will be easier to adopt, maintain and update. This chapter also introduces the first, to the best of our knowledge, formulation of an analysis that determines a GEMM matrix's access order by matching the induction variables used in the memory access (see Section 3.2.1). Access-order detection is crucial to enable transparent and correct usage of high-performance libraries.

This chapter makes the following contributions:

1. A robust idiom-recognition compiler pass that identifies many variants of the GEMM and SYR2K idioms and replaces them with optimized library calls (see Section 3.2 and Section 3.2.3). **Kernel Finder & Replacer (KernelFaRer)**[1] identifies both näive implementations and hand-optimized variants of these idioms(see Section 3.3.3).

2. The first formulation of an analysis that determines a GEMM matrix's access orders by combining pattern matching and loop information analysis in LLVM **Intermidiate Representation (IR)** (see Section 3.2.1). The same strategy was employed to match SYR2K's access order.

3. An experimental evaluation that provides evidence that: (a) the pass is robust and identifies many more idiom variants than other approaches; (b) the addition to compilation time is significantly smaller; (c) the increase in performance is consistent across architectures and libraries (see Section 4.2).

The remainder of the chapter is organized as follows: Section 3.1 presents

[1]Source code available at `https://github.com/jaopaulolc/KernelFaRer`.

background material; Section 3.2 details how LLVM IR pattern matching and optimized library call insertion are combined in KernelFaRer's implementation; Section 4.2 describes the experimental setup and analyzes the performance improvements achieved by the proposed approach when compared with current solutions and manual library-based programming.

3.1 Pattern Matching Idioms

This section presents programming idioms in the context of pattern matchers that target idioms in LLVM IR code and reviews how the GEMM operation is typically programmed and optimized.

3.1.1 Programming Idioms

Programming idioms are recurrent constructs that express a computation, can be easily recognized (by humans), and are simple to compose [74]. The introduction of the array-oriented APL language in the 1960s [54] with concise statements that exhibit high memory and high runtime complexity motivated research on idiom recognition and selection [82]. Later, recognition of idioms was used in many programming languages [14], [18], [34], [41], [44], [45], [60], [68], [75]. Existing idiom-based approaches differ mostly on:

- the program representation — usually graph-based structures (*e.g.* data-dependence graphs, expression trees);
- the matching algorithm (*e.g.* depth-first traversals or solver-based);
- the normalization constraints for the matching mechanism to work (*e.g.* memory accesses must be affine) or to be more effective (*e.g.* common subexpressions elimination);
- how idioms are expressed, either through a domain-specific language (DSL) or programatically via a Visitor Pattern [72], and replaced (*e.g.* code generation or employment of high-performance library).

```
for (i = 0; s[i]; i++);
```
(a)

```
(i < j) ? i : j;
```
(b)

Figure 3.1: (a) Idiom of finding the length of a string and (b) returning the minimum of 2 numbers.

Idioms have concise syntax and convey common understanding of recurrent computations [54]. For example, Figure 3.1(a) shows an idiom, written in C, for finding the length of a string `s`. This example relies on the `NULL` byte at the end of well-formed strings, a false value in C that terminates the loop. Figure 3.1(b) shows how a ternary operator concisely expresses the `min` operation without the need for `if/else` control flow. This chapter identifies larger idioms that express an identifiable and well-known operation: GEMM. These idioms often account for a significant portion of the execution time of an application and their performance can be greatly improved by replacing them with calls to fine-tuned libraries.

3.1.2 Pattern Matching in LLVM IR

The LLVM compiler framework has a `PatternMatch` namespace that provides a mechanism to build simple and efficient matchers for LLVM IR [64]. `PatternMatch` is used for static analysis (*e.g.* Demanded-Bits and Instruction-Simplify analyses), code generation (*e.g.* Instruction Selection) and transformations (*e.g.* Instruction Combining). The code in Figure 3.2 illustrates how `PatternMatch` can simplify instructions in the LLVM `InstCombine` pass.

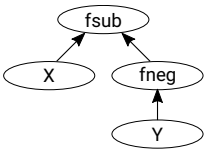
Figure 3.2(b) shows the tree representation of the expression in the return statement in line 3 of Figure 3.2(a). Figure 3.2(c) has a code excerpt from the LLVM `InstCombine` pass that employs `PatternMatch` to simplify the expression $X - (-Y)$ into $X + Y$, shown as a tree in Figure 3.2(d) [2]. In the code of Figure 3.2(c), `I` references an `fsub` (floating-point subtract) instruction. The method `match(V, P)` returns true if and only if the value V matches the pattern P.

`PatternMatch` provides template methods that both describe IR patterns

[2]The code in LLVM uses a visitor for `fsub` instructions that only matches `fneg`.

```
1 float foo (float X, float Y)
2 {
3   return X - (-Y);
4 }
```

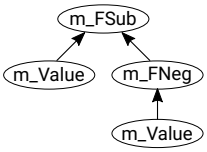
(a)



(b)

```
1 // X - (-Y) => X + Y
2 if (match(I,
3           m_FSub(
4             m_Value(X),
5             m_FNeg(
6               m_Value(Y)))))
7 {
8   return CreateFAdd(X, Y, &I);
9 }
```

(c)



(d)

Figure 3.2: (a) Double negation example; (b) Tree representation of (a); (c) Matcher and replacement code; and (d) Pattern matched.

and bind values in the pattern to pass variables. For example, the `m_FSub` and `m_FNeg` methods shown describe the instructions `fsub` and `fneg`. The `m_Value` method matches and binds the left-hand side operand of `fsub` to X and the single argument of `fneg` to Y. `PatternMatch` also provides flexible commutative versions of matcher methods (*e.g.*`m_c_FAdd`[3]) that avoids cumbersome code describing and testing two symmetric patterns. Within a basic block, LLVM can find simple patterns used for instruction selection.

The LLVM `LoopIdiomRecognize` attempts to recognize two idioms — memory-set and memory-copy — to replace the loop code with the `llvm.memset` and `llvm.memcpy` intrinsics. Backends generate inline code or runtime library calls for these intrinsics. The existing LLVM `PatternMatch` cannot be used in the `LoopIdiomRecognize` pass because it is limited to the scope of a single basic block. Instead, that pass employs only **Data-Flow Analyses (DFAs)** [1] to identify and replace both memory idioms. KernelFaRer uses `PatternMatch` to find loop idioms and DFAs to assess if the the transformations are legal.

3.1.3 General Matrix-Matrix Multiplication

General matrix-matrix multiplication (GEMM), already pervasive in linear algebra computations [46], regained attention because convolution kernels

[3] `_c` indicates that the instruction to be matched (`fadd`) is commutative.

```
1  void cblas_dgemm (
2    const CBLAS_LAYOUT Layout,
3    const CBLAS_TRANSPOSE transa,
4    const CBLAS_TRANSPOSE transb,
5    const int m, const int n, const int d,
6    const double alpha,
7    const double    *a, // A's base address
8    const int      lda, // and leading dimension
9    const double    *b, // B's base address
10   const int      ldb, // and leading dimension
11   const double  beta,
12   double          *c, // C's base address
13   const int     ldc); // and leading dimension
```

Figure 3.3: CBLAS interface for double-precision GEMM.

used in neural networks can be efficiently implemented using GEMM as a primitive [19]. Formally, GEMM can be defined as follows.

Definition 1. Let A and B be matrices of dimensions $M \times D$ and $D \times N$ respectively. Let α and β be any value in $\mathbb{R}$. The general matrix-matrix product of A and B is a matrix C of dimensions $M \times D$ such that:

$$C(i,j) = \beta \cdot C(i,j) + \alpha \cdot \sum_{k=1}^{D} A(i,k) \cdot B(k,j) \tag{3.1}$$

Although a naïve implementation of GEMM is simple, seminal work by Goto et al. shoroughly 80%ws that a fast GEMM must utilize the memory hierarchy well [35]. Their approach splits the computation into blocks to increase data reuse; and creates a matrix memory layout that increases temporal and spatial access locality, a process called *packing the matrices*. The central idea is to focus on the data movement from main memory, through caches, and into the processor's registers.

The GotoBLAS library [35] inspired other BLAS libraries. Processor-manufacturer solutions include IBM's **Engineering and Scientific Subroutine Library (ESSL)** [49] and Intel's **Math Kernel Library (MKL)** [51]. Manufacturers also contribute to OpenBLAS [70]. AMD's version is the **BLAS-like Library Instantiation Software (BLIS)** framework [94]. CBLAS is a unified interface to these libraries for C languages [9].

Figure 3.3 shows the interface for double-precision GEMM. The `Layout` specifies if the resulting matrix C is stored in row-major or column-major order. The arguments `transa` and `transb` indicate if the matrices A and B

are transposed, allowing matrices in different storage orders to be multiplied without the creation of copies. An idiom matcher (see Section 3.2) must deduce the layout of the matrices being matched to obtain these first three parameters. The matrix dimensions (`m`, `n`, and `d`) appear in line 5, followed by the scalar factors α and β (lines 6 and 11), and the base address pointers (lines 7, 9, and 12). The leading dimensions (lines 8, 10, and 13) are the number of elements in the *first* dimension of each matrix.

BLAS libraries rely heavily on direct use of assembly and thus are not portable across platforms, thus the many versions of BLAS. In contrast, Eigen [38] implements BLAS routines using C++ template meta programming to hierarchically organize the decisions and the tuning for each platform. The templates encode platform-specific information that allow the compiler to choose a strategy that is tuned based on cache sizes, vectorization support and other instruction-set-architecture features (*e.g.* fused multiply-add support). Eigen also relies on code transformations available in modern compilers such as **GNU's C/C++ Compiler (GCC)** and Clang [21]. There is no routine to call to compute GEMM in Eigen because it is a header-file only library that defines builtin vector and matrix types. Computations are overloaded operations on these types. Thus, the programmer needs to include the header files providing Eigen's types and operators and write C++ expressions of the form: `C = beta * C + alpha * A * B` as per Definition 1. Using Eigen to replace a GEMM idiom required the creation of a wrapper library encapsulating Eigen code (see Section 3.2.3).

3.2 An Idiom Recognition and Replacement Pass

This section presents KernelFaRer, an LLVM IR pass that performs idiomatic code rewrite and is integrated into LLVM's standard optimization pipeline. KernelFaRer is an independent IR pass that extends the LLVM `PatternMatch` **Application Programming Interface (API)** (see Section 3.1.2) with custom matchers to identify more complex idioms. KernelFaRer works

independently of other LLVM optimization frameworks like Polly, but it can be of assistance to it or other paths to code generation. The description of this pattern-matching extension is demonstrated in a GEMM case study. However, the methodology is flexible enough to capture all IR constructs and thus enables the description of many idioms. In fact, performance results with our prototype implementation of a SYR2K matcher and replacer are discussed in Section 4.2. A standard data-flow analysis is sufficient to determine if values computed in the idiom are used elsewhere in the program and thus can also be extended to other idioms.

The KernelFaRer's algorithm can be divided into three phases as follows:

1. Identify candidates that match the target idiom through IR matchers (see Section 3.2.1).

2. Check data dependences and isolate the matched code (see Section 3.2.2).

3. Replace the idiom with a call to a high-performance library (see Section 3.2.3).

Phase 1 uses LLVM's `PatternMatch` to identify IR code that matches the target idiom. The data-dependence analysis in Phase 2 determines if the replacement of the matched code with a library call is legal. This phase also determines if code transformations, such as loop distribution or loop invariant code motion, are needed to make the transformation legal.

3.2.1 GEMM Pattern Matching (Phase 1)

GEMM memory access patterns can be expressed in a higher-level programming language as shown in Figure 3.4. The variables iv_i, iv_j, and iv_k on line 1 of Figure 3.4(a) are induction variables of a level-three loop nest (or deeper). The `for` loop syntax in Figure 3.4(a) indicates that the three loops can be nested in any order. For any nesting order, the reduction on line 2 expresses a GEMM. Parenthesis operators are used for indexing the arrays to indicate that the elements of the array may be accessed either in column-major or in row-major order. For instance, $A(i, k)$ means that the element on the i-th

```
1 for (0 <= iv_i < M; 0 <= iv_j < N;
2       0 <= iv_k < D)
3   C(iv_i,iv_j) += A(iv_i,iv_k) * B(iv_k,iv_j)
```

(a)

```
1 operator()(iv_1, iv_2) {
2   return *(addr + iv_2 * ld + iv_1);
3 }
```

(b)

```
1 operator()(iv_1, iv_2) {
2   return *(addr + iv_1 * ld + iv_2);
3 }
```

(c)

```
1 %41 = phi [%53, %40], [0, %25]
2 %42 = phi [%52, %40], [0, %25]
3 %43 = mul %41, %17
4 %44 = add %43, %23
5 %45 = getelementptr %4, %44
6 %46 = load %45
7 %47 = mul %41, %16
8 %48 = add %47, %26
9 %49 = getelementptr %6, %48
10 %50 = load %49
11 %51 = fmul %46, %50
12 %52 = fadd %42, %51
13 %53 = add %41, 1
14 %55 = icmp eq %53, %21
15 br %55, %30, %40
```

(d)

Figure 3.4: (a) Memory access of GEMM in source code; (b) colum-major access order; (c) row-major access order; (d) Simplified LLVM IR code of the innermost loop in (a) (Code in (b) and (c) is in C/C++).

row and k-th column is accessed. The implementation of the operators for column-major order is shown in Figure 3.4(b) while the row-major access is shown in Figure 3.4(c), where `addr` is the base address of an array and `ld` is the leading dimension of the array. The dimensions of the matrices are: $A_{M\times D}$, $B_{D\times N}$, and $C_{M\times N}$.

Identifying a GEMM idiom requires the identification of its two components: the loop nest on line 1 and the reduction operation on line 2 of Figure 3.4(a). The multiply-and-add operations of the reduction idiom appear on lines 11-12 of the LLVM IR of the innermost loop nest shown in Figure 3.4(d). In this **Static Single Assignment (SSA)** representation the result of each instruction is assigned a unique value. Thus, each of the instructions in Figure 3.4(d) can be referred to by its value `V`. For instance, the `fmul` in line 11 is uniquely identified by the value `%51` and the `fadd` in line 12 by the value `%52`. In general, when presenting an algorithm that iterates over all the instructions in the body of a loop `L`, this chapter will say "`for` all values `V` in `L`".

A *GEMMReduction* is a multiply-add instruction sequence that satisfies the following conditions: 1. it appears in the innermost level of a loop nest of at least depth three; 2. the operands are memory accesses to arrays; 3. the address

```
template <typename MatcherType>
auto MatchStoreOfMatrixC(MatcherType GEMMReduction, Value C, Value
    Alpha, Value Beta, PHINode iv1, PHINode iv2, Value LDC, Value GEP)
     {
  return m_Store(
    OneOf(ScaledVOrV(Alpha, GEMMReduction),
          ScaledVOrV(Beta, m_PHI(m_Value(), GEMMReduction)),
          m_c_FAdd(ScaledVOrV(Beta, m_Load(GEP)),
                   ScaledVOrV(Alpha, m_PHI(m_Value(), GEMMReduction)
                       ))),
    ArrayAccess(C, iv1, iv2, LDC));
}
```

(a)

```
inline auto GEMMReduction(Value AddLHS, Value MulLHS, Value MulRHS,
    PHINode iv1A, PHINode iv2A, PHINode iv1B, PHINode iv2B, Value Alpha,
    Value LDA, Value LDB) {
  return MultiplyAdd(
      Alpha, AddLHS, m_Load(ArrayAccess(MulLHS, iv1A, iv2A, LDA)),
      m_Load(ArrayAccess(MulRHS, iv1B, iv2B, LDB));
}
```

(b)

```
auto ArrayAccess(Value Op, PHINode iv1, PHINode iv2, Value LD) {
  return OneOf(
      m_GEP(m_Load(m_GEP(Op, m_PHI(iv2))), m_PHI(iv1)),
      m_GEP(m_GEP(Op, PHITimesLD(iv1, LD)), m_PHI(iv2)),
      m_GEP(m_GEP(Op, m_PHI(iv2)), PHITimesLD(iv1, LD)),
      m_GEP(Op, m_PHI(iv1), m_PHI(iv2)),
      m_GEP(Op, AffineFunctionOfPHI(iv1, iv2, LD));
}
```

(c)

Figure 3.5: (a) Matcher of a store of *GEMMReduction* into matrix C; (b) *GEMMReduction* matcher; and (c) Matcher for an array access.

of the memory accesses are affine expressions of the form `addr`$+iv_x\times$`ld`$+iv_y$, where iv_x and iv_y are index variables in the loop nest and `addr` is a loop-invariant expression — either the base address of a matrix or the base address or a block within the matrix; and 4. the combination of induction variables used in the address expressions is one of the combinations shown in Table 3.1.

The first component of the pattern, loop nests, are identified using LLVM's LoopInfo analysis pass. LoopInfo provides a consistent way to retrieve loop information from the IR of a program, such as the nesting level of a given loop. The second part of the pattern, *GEMMReduction*, cannot be identified with the existing LLVM `PatternMatch` API. Therefore, KernelFaRer extends `PatternMatch` by adding new matchers and new constructs for matching more complex patterns. Figure 3.5 shows the main patterns contributed by

KernelFaRer, those printed in red are the proposed extensions and those in black are the existing constructs. The main pattern that matches a store of a *GEMMReduction* to the destination matrix C is shown in Figure 3.5(a), a similar pattern was created to match SYR2K reductions.

KernelFaRer introduces the `OneOf` combiner construct to allow a list of multiple matchers, which usually represent subtle variations in a target pattern, to be tested in turn. `OneOf` only returns a successful match if one of the provided sub-matchers match the underlying piece of LLVM IR. This construct is widely used in KernelFaRer extensions to facilitate the description of variations in the target pattern. `PatternMatch` provides basic disjunctive and conjunctive nodes. Disjunction nodes allows the capturing of idioms with polymorphic components. Conjunction nodes allow further specification and pattern component constraints. Combining `OneOf` with these nodes makes KernelFaRer pattern matcher more robust. For example, $ScaledVOrV(s, V)$ uses the disjunctive node to match either $s * V$ or V, where V is any value represented by a KernelFaRer & `PatternMatch` matcher.

Figure 3.5(b) shows the matcher that captures the *GEMMReduction* itself, where `MultiplyAdd` matches expressions of the form $\alpha * A * B$.[4] `ArrayAccess`, shown in Figure 3.5(c), is the matcher that identifies accesses to arrays in different representations. Lines 3-6 are the sub-matchers representing different variants of access to 2D arrays, while in line 7 the matcher identifies flat-arrays. `m_GEP` and `m_PHI` match their respective IR instructions, namely `getelementptr` and ϕ`-nodes`. Flat-arrays have indexing expressions that are affine functions of loop induction variables and such expressions are matched with `AffineFunctionOfPHI`. 2D-array accesses have an idiomatic expression that multiplies an induction variable by a matrix leading dimension. `PHITimesLD` captures two variants of this idiom, thus matching wether the multiplication is performed by a multiply (`mul`) or by a shift-left (`shl`) LLVM IR instruction.

[4] α is optionally matched by using `PatternMatch`'s disjunctive node.

Algorithm 1 LLVM IR Pass to find GEMM candidates.

```
1: function findGemmIRPass(Function F, LoopInfo LI)
2:   LoopList ← findInnerDeepLoops(F, LI)
3:   for all Loops L in LoopList do
4:     for all Values V in L do
5:       if GemmPattern.Match(V) then
6:         IVList ← {iv_1^A, iv_2^A, iv_1^B, iv_2^B, iv_1^C, iv_2^C}
7:         (OrderList, iv_i, iv_j, iv_k) ← MatrixAccessOrder(IVList)
8:         (M, N, D) ← LoopsUpperBound(LI, iv_i, iv_j, iv_k)
9:         if AllGemmValuesFound() then
10:          A ← Matrix(OrderList, M, N, D)
11:          B ← Matrix(OrderList, M, N, D)
12:          C ← Matrix(OrderList, M, N, D)
13:          Gemm ← (L, IVList, A, B, C)
14:          Candidates.insert(Gemm)
15: function match(Value V, Pattern P)
16:   return P.match(V)
17: interface Pattern<T>::match(Value V)
18: function Pattern<FAdd>::match(Value V)
19:   AddOper0 ← V.getOperand(0)
20:   AddOper1 ← V.getOperand(1)
21:   return FAdd_Match(AddOper0, AddOper1)
```

Algorithm to match a GEMM

The central idea of this chapter is that the loop information provided by the compiler can be used to constrain the search for the target idiom's components to the places where it is possible for them to occur. In LLVM, the nesting level, entry, exit, and latch basic blocks for each loop are available through the `LoopInfo` pass. However, there are two limitations: (1) in a kernel that has been optimized through blocking, the inner loops are not in canonical form; and (2) the induction-variable information is only available for canonical loops via the `LoopInfo` pass [64]. KernelFaRer's solution is to combine basic-block and loop-nesting information from LoopInfo with `PatternMatch`. In LLVM IR, induction variables are lowered to ϕ instructions with at least two incoming values: an initialization value; and a new value that comes from the loop's latch basic block. KernelFaRer's general idiom recognizer uses `PatternMatch` to match these ϕ instructions.

Algorithm 1 uses `PatternMatch` and the `LoopInfo` pass to identify GEMM

Table 3.1: Access offset expressions for all combinations of column-major (CM) and row-major (RM) order.

Access Order			Offset Expressions		
C	A	B	C	A	B
RM	RM	RM	$iv_i \times ld_C + iv_j$	$iv_i \times ld_A + iv_k$	$iv_k \times ld_B + iv_j$
CM			$iv_j \times ld_C + iv_i$		
RM	CM		$iv_i \times ld_C + iv_j$	$iv_k \times ld_A + iv_i$	
CM			$iv_j \times ld_C + iv_i$		
RM	RM	CM	$iv_i \times ld_C + iv_j$	$iv_i \times ld_A + iv_k$	$iv_j \times ld_B + iv_k$
CM			$iv_j \times ld_C + iv_i$		
RM	CM		$iv_i \times ld_C + iv_j$	$iv_k \times ld_A + iv_i$	
CM			$iv_j \times ld_C + iv_i$		

candidates. First, using the LLVM `LoopInfo` pass, find all innermost loops that are nested at the third (or deeper) level and place these loops in a list (line 2). Then, iterate over all instructions (value V in line 4) inside those loops, invoking the method MATCH of a `GemmPattern` object on each one of them. The generic interface for MATCH is shown in line 17, where `T` can be any LLVM IR instruction type. Each instruction type in a pattern must have an implementation for the MATCH interface. For instance, the implementation of MATCH for `FAdd` for the GEMM pattern is shown on line 18[5]. To find the idiom, MATCH descends the IR tree matching the specified pattern.

Values from the IR are captured through matcher objects of type BINDTYPE. `GemmPattern` contains such objects to capture, for instance, values associated with a GEMM's induction variables and memory address instructions.

Determination of the Matrix Access Order

A crucial step to replace a GEMM idiom with a call to a library is to determine the access order used by the idiom for each matrix (Algorithm 1, line 7). These access orders are required by the library call that will replace the detected idiom (see Section 3.2.3). This analysis differs from the work of Wolfe et al. where the iteration domain is analyzed to identify dependencies in a loop [71].

[5]The FADD_MATCH in line 21 is an algorithmic simplification of an object-oriented code structure in LLVM. It abstracts the matching of both operands of a `fadd` instruction referenced by `V` (line 18) using the `FAdd_Match` object within `GemmPattern`.

Table 3.2: Conditions to determine the access orientation.

Access Order		Equalities to derive access order of		
A	B	A and B	C	
			RM	CM
RM	RM	$o_A = l_B$	$l_C = l_A \wedge o_C = o_B$	$l_C = o_B \wedge o_C = l_A$
CM	RM	$l_A = l_B$	$l_C = o_A \wedge o_C = o_B$	$l_C = o_B \wedge o_C = o_A$
RM	CM	$o_A = o_B$	$l_C = l_A \wedge o_C = l_B$	$l_C = l_B \wedge o_C = l_A$
CM	CM	$l_A = o_B$	$l_C = o_A \wedge o_C = l_B$	$l_C = l_B \wedge o_C = o_A$

Here the goal is to identify the access order of each matrix in the target pattern.

Table 3.1 shows the offset expressions according to the access order of the matrices assuming that the index variables are iv_i, iv_j, and iv_k. When a multiply-and-add operation that is a candidate to be a GEMM reduction is encountered, the expressions for the access into the matrices are identified as $l_C \times ld_C + o_C$, $l_A \times ld_A + o_A$, and $l_B \times ld_B + o_B$ where l_C is the index variable for the leading dimension of matrix C and o_C is the offset into that dimension of C[6]. The same logic is applied to the matrices A and B. The access order can be deduced by examining which of the index variables are identical in the multiply-and-add idiom candidate.

Table 3.2 shows the equality conditions that determine the access order for matrices A and B. For instance, if $o_A = l_B$, then both matrices A and B are accessed in row-major order as shown in the first two rows of Table 3.1. Given that all the possible combinations for access orders for a GEMM are given in Table 3.1, it follows that if none of the equality constraints shown on the second column of Table 3.2 are met, then the multiply-and-add cannot be a GEMM reduction.

To illustrate how to determine the access order of C, let's examine the two top rows of Table 3.1 where both matrices A and B are accessed in row-major order. There are two cases. If $l_C = l_A$ and $o_C = o_B$ then C is accessed in row-major order. If $l_C = o_B$ and $o_C = l_A$, then C is accessed in column-major order. If neither of these conjunctions are satisfied, then the multiply-and-add

[6] l_X and o_X are also known as the angular and linear coefficients of a linear function, such as those in indexing expressions used to access array elements.

is not a GEMM reduction. Similar sets of conjunctions can be written for the other three combinations of access order for A and B as shown in Table 3.2.

This strategy can also be used to detect the access order of matrices in other computational kernels (*e.g.*, SYR2K).

Loop Upper Bounds

The intuition to determine loop upper bounds is that the index variable of a loop must be initialized prior to starting the execution of the loop and must be updated within the loop body. Therefore, in the SSA representation there must be a ϕ instruction that merges the initialization path with the update path at the first basic block of a loop. First, add all ϕ instructions that are in the first basic-block of the loops and that use iv_i, iv_j, or iv_k to a work list. Then, match each ϕ instruction in this list against a pattern tree that represents the instruction sequence for the loop index update and loop comparison[7]. The upper bound of a loop L is the operand compared with the value defined by the ϕ instruction if the pattern belongs to the latch block of L. All GEMM instances matched by `GemmPattern` that have a valid access order and for which loop nests upper bounds can be determined are candidates for replacement with a call to a library (see Section 3.2.3). Now, a data-flow analysis must establish the legality of the transformation.

In the triangular iteration space of SYR2K two out of tree induction variables are traversed in the same way as in GEMM and the third variable's (J) upper (lower) bound is dependent on another variable (I). To detect this pattern, KernelFaRer checks for the presence of I in the ϕ instruction for J. This detection method can be thwarted by the insertion of auxiliary induction variables.

3.2.2 Data-Dependence Analysis

Idiom replacement must preserve program behavior, including writes to memory and outputs. Also, any inner-loop code that is not part of the GEMM reduction must be safely moved out of the loop nest. A liveness and side-effects analysis

[7]This instruction sequence is idiomatic of loop-exit conditions.

Algorithm 2 Data-Dependence Analysis IR Pass.

```
 1: function AnalysisPass(Function F, Kernel K)
 2:   L ← K.getAssociatedLoop()
 3:   for all Instructions I in L do
 4:     if I ∈ K.Values or I ∈ K.Stores then continue
 5:     if I.mayWriteToMemory() then
 6:       return False
 7:     if I.mayHaveSideEffects() then
 8:       return False
 9:     for all Users U of Instruction I do
10:       B ← BasicBlock of I
11:       if B ∉ L then
12:         return False
13:     return True
```

of the use-def chains in the idiom's loop nest can be combined to determine if any intermediate value is live after the idiom or produces side-effects during the computation. This analysis differs from the loop dependence analysis of Wolfe et al. [71].

The Data-Dependence Analysis in Algorithm 2 receives the IR for a function F and a Kernel object found within F. It returns a *boolean* indicating if the idiom replacement is legal. At the matching stage, the algorithm gathers two data structures. ($K.Values$) contains matrix pointers, offsets and intermediate values; ($K.Stores$) contains the stores in the Kernel. These data structures afford flexible access to the Kernel parameters and enable checking that the only extra stores in the loop nest are initializations stores.

The Data-Dependence Analysis algorithm iterates over all instructions in the kernel's associated loop (line 3) skipping those that are in the kernel (line 4). If an instruction may store to memory (line 5) or produces side-effects such as throwing an exception (line 7), return *False*. The algorithm also checks for data flow from definitions in the kernel innermost loop to uses after the loop (lines 9 - 12).

The kernels in Figure 3.6 illustrate when instructions may or may not be moved out of a GEMM loop. Two of them, Figure 3.6(b) and (c), are from benchmarks in PolyBench [77]. In spite of its simplicity, the naïve GEMM implementation in Figure 3.6(a) cannot be directly rewritten because of the

```
1 for (long i = 0; i < M; i++)
2   for (long j = 0; j < N; j++) {
3     C[i][j] = 0.0;
4     for (long k = 0; k < D; k++)
5       C[i][j] +=
6         alpha * A[i][k] * B[k][j];
7   }
```

(a)

```
1 for (long i = 0; i < M; i++)
2   for (long j = 0; j < M; j++)
3     for (long k = 0; k < N; k++) {
4       C[i][j] +=
5         alpha * A[i][k] * B[j][k];
6       C[i][j] +=
7         alpha * B[i][k] * A[j][k];
8     }
```

(b)

```
1 for (long r = 0; r < R; r++)
2   for (long q = 0; q < Q; q++) {
3     for (long p = 0; p < P; p++) {
4       sum[r][q][p] = 0.0;
5       for (long s = 0; s < P; k++)
6         sum[r][q][p] +=
7           A[r][q][s] * C4[s][p];
8     }
9     for (long p = 0; p < P; p++)
10      A[r][q][p] = sum[r][q][p];
11  }
```

(c)

Figure 3.6: (a) Naïve GEMM; (b) symmetric rank-2k operations (syr2k); and (c) Multiresolution analysis kernel (doitgen).

initialization of matrix C. This initialization must first be moved to a separate loop because all high-performance libraries assume that matrix C is initialized prior to calling the function that implements GEMM. Another possibility is to simply delete the GEMM reduction store that is in line 5 and to insert a library call at the exit-block of the loop nest — dead code can be removed later by LLVM passes. Side effects in the initialization of the matrix C could also prevent rewrite. KernelFaRer recognizes this idiom and replaces it if no side-effects exist.

Figure 3.6(b) shows a symmetric rank-2k operation from the SYR2K kernel. This kernel accesses matrix A in row-major order and B in column-major order in line 4 and then matrix B in row-major order and A in column-major order in line 5. C is accessed in row-major order. The replacement of the idioms in SYR2K is safe and the access order is detected by the algorithm. No copy operations are required because a matrix stored in row-major order can be accessed in column-major order by specifying to the library that the matrix is transposed. However, floating-point arithmetic is not associative and, depending on the values in A and B, the result after and before the transformation might differ for a given error tolerance. In this case there are three options: 1. do not replace it; 2. replace it with two calls to `cblas_gemm` if the the user allows it; 3. replace it with a call to `cblas_syr2k`. KernelFaRer replaces this idiom with a call to `cblas_syr2k`.

The multiresolution analysis `doitgen` kernel shown in Figure 3.6(c) has `R` GEMM instances, one for each resolution `r`. The GEMM in each resolution can only be rewritten after the update of row `q` in A with the values from `sum` (lines 8-9) is moved to a new loop-Q after the original one (line 2). Loop distribution is not currently available in KernelFaRer because LLVM only distributes innermost loops.

3.2.3 Idiom Rewrite

The prototype of KernelFaRer evaluates two replacement options: a custom library that leverages Eigen [38] or the CBLAS [9] interface, which is widely used by OpenBLAS [93] and by vendor-specific libraries such as MKL [51], ESSL [49] and BLIS[8] [94]. This interface allows the evaluation of multiple libraries without recompiling the program. CBLAS expects that matrices be stored contiguously in memory, thus KernelFaRer does not replace any GEMM that access the matrices through double pointer indirection, *e.g.*`float**`, because the contiguity constraint cannot be proven. In addition, instaces where only sub-blocks of 2D arrays are accessed to compute GEMM are also not replaced since the contiguity constrain is violated.

Unlike BLAS, Eigen [38] uses a minimum amount of assembly code; is portable; and supports all basic types from C/C++ languages, not just single and double-precision floating-point types. KernelFaRer creates a slim wrapper routine for each data type in Eigen. Eigen heavily exploits C++ template meta programming and other object-oriented features (*e.g.* polymorphism). In Eigen's template type hierarchy the compiler chooses which methods to instantiate based on the target platform (see Section 3.1.3).

The following steps replace a GEMM-idiom loop nest with a library call: 1. insert a call instruction in the exit-block of the idiom's loop nest. The values captured in the matching phase (Section 3.2.1) are passed as arguments to the library call; 2. delete the idiom's store to the resulting matrix. This makes the computations in the idiom's loop nest dead-code, which can be removed by

[8]BLIS is actually an open-source project but AMD provides support and advertises it as their platform-specific solution: `https://developer.amd.com/amd-aocl/blas-library/`

Table 3.3: Machine configuration used in the evaluation.

Component/Feature	Intel®	IBM®	AMD®
Processor	2x Xeon™ 8268	2x Power9™	1x EPYC™ 7742
Cores/Threads	24/2	20/4	64/2
L1 instruction cache	32KiB	32KiB	32KiB
L1 data cache	32KiB	32KiB	32KiB
L2 (unified)	256KiB	512KiB	512KiB
L3 (unified+shared)	35.75MiB	10MiB	16MiB
RAM	755GiB	1TiB	995GiB
Memory bandwidth	131.13GiB/s	140GiB/s	190.7GiB/s
FMA Latency	4	7	5
FMA Throughput	2	2	2

LLVM passes scheduled afterwards. 3. run LLVM's loop deletion, dead-code elimination and CFG simplification passes.

3.3 Experimental Evaluation

This experimental evaluation assesses 1. the effect on performance of replacing native code idioms with high-performance library calls on three platforms: Intel x86, AMD x86, and IBM PowerPC (Section 3.3.2); 2. the robustness of KernelFaRer's pattern matching in comparison with Polly and IDL [34] (Section 3.3.3); and 3. the effects of pattern matching and analysis on compilation times (Section 3.3.5).

3.3.1 Experimental Setup

This section presents the detailed experimental setup, methodology, and all tools used in the experimental evaluation presented next.

Machine Setup

This experimental evaluation uses the three platforms shown in Table 5.1 to determine if the performance trends are platform specific [9][10]. All platforms run Linux with a 64-bit kernel at version `4.15.0-115-generic`. The processor's frequency was locked to 3.5 GHz on the Intel machine and to 2.3 GHz on the IBM machine. Issues with the `acpi-cpufreq` module prevent the locking of

[9]Core counts are per socket and thread counts are per core.
[10]Cache sizes are per core or per core pair.

the frequency on the AMD machine and thus the frequency fluctuates between 1.8 GHz and 2.2 GHz.

Compilation Pipeline

All benchmarks are compiled using Clang and LLVM tools from the proprietary LLVM 12.x branch at `-O3` with `-mtune=native` and the options `-march=native` for x86 (Intel and AMD) and `-mcpu=native` for IBM PowerPC. IDL [34] and KernelFaRer need to run prior to loop unrolling and vectorization passes. The solution is to disable these passes via `-fno-unroll-loops -fno-vectorize -fno-slp-vectorize` before the idiom-identification pass and to reenable them afterwards by running the `opt` LLVM tool using `-O3`. All binaries, including the baselines, were compiled with `-ffp-contract=fast` and `-ffast-math` for a fair comparison with libraries that assume a relaxed floating-point contract.

Backends

This evaluation uses the following backends: BLAS (via the OpenBLAS implementation [70], [93]), Eigen [38], MKL [51], BLIS [94], and ESSL [49]. It uses the latest stable release, from source, of BLAS[11] and Eigen[12]. It uses the most recent, pre-built, binaries available from the vendors for the platform-specific backends — MKL[13], BLIS[14], and ESSL[15].

Libraries are unmodified except for setting flags to enable multithreading. The number of threads in each platform is chosen such that all computation is performed on a single socket and no hyperthreading takes place. This criteria results in 24 threads for the Intel platform, 64 for the AMD, and 20 for the PowerPC (see Table 5.1). The same number of threads is used in each platform for all backends.

[11] Default branch at 1f62a8278983f7afec8c9c28ecbb2f4892f7ce52.
[12] Default branch at b5df8cabd7b9dcaf3eb0ab93416f3f25352c55f2.
[13] Version 2020.0 build 20191122.
[14] Version 1.3 build 20190901.
[15] Version 6.2.1.

Polly

This evaluation uses the version of Polly available in the proprietary LLVM 12.x branch. The target-specific instruction scheduling in Polly's GEMM optimization pass requires the following platform information: size and associativity of first and second level caches; latency and throughput of vector fused multiply-add instruction (FMA). This information was extracted from each vendor's software optimization guides [3], [48] and Agner Fog's instruction tables [26] and included the architecture-specific information in Table 5.1 For the evaluation, Polly's code uses the same thread counts as the other approaches. Even though pattern-matching options are always enabled in Polly, it is only successful in four out of seven programs (see Section 3.3.3).

IDL

IDL [34] does not provide a replacement pass (see Section 3.2.3). This evaluation focuses on the idiom identification mechanism of IDL. The performance improvement of replacing idioms identified by IDL should be the same as when using KernelFaRer. Normally, IDL also has the ability to detect multiple idioms at once, but in this evaluation, all other idioms have been disabled, leaving only GEMM.

PolyBench

This evaluation uses all of the benchmarks in the PolyBench/C 4.2 benchmark suite [77], a set of benchmarks originally curated for the testing of Polly. Choosing the extra-large datasets ensures that the input and output matrices do not fit in cache, thus allowing assessment of the cache and memory-hierarchy awareness of each evaluated library and strategy. This study evaluated all the benchmarks in the suite but presents results only for benchmarks where at least one of the approaches recognized a GEMM pattern. Section 3.3.2 discusses the impact on the other benchmarks.

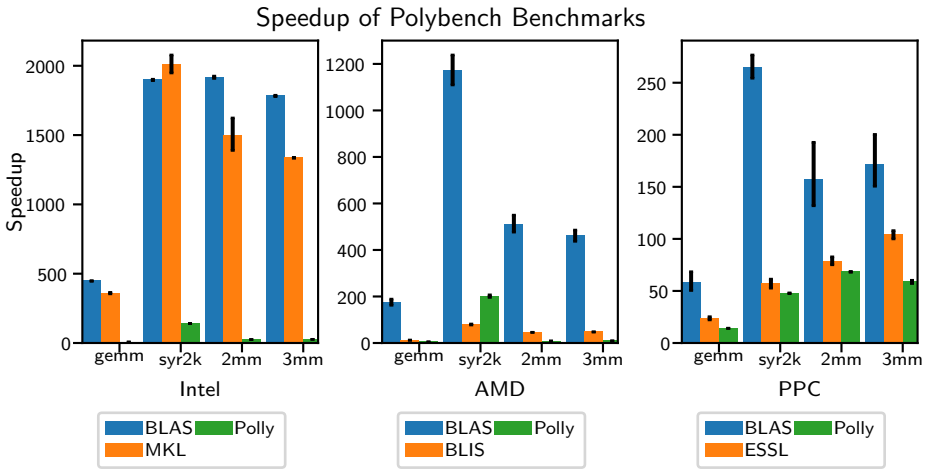


Figure 3.7: The speedup of benchmarks when compared to the same benchmark run at `-O3` on the respective platform.

Experimental Methodology

The methodology proceeds as follows. 1. Compile each benchmark, for each platform, with aggressive optimization flags (see Section 3.3.1) using five strategies: baseline, Polly, and replacement with BLAS, Eigen, and a platform-specific library (MKL for Intel, BLIS for AMD, and ESSL for PowerPC). 2. Create a list containing all the executables. 3. Measure the execution time of each element of the list once. 4. Randomize the list of executables before the next set of measurements. 5. Repeat until there are twenty measurements for each executable. This methodology ensures that changes to the execution environment that may affect performance manifest in a higher variance between executions of the same executable rather than introduce bias in the results of the experimental evaluation. All speedups are relative to the platform-specific, baseline application code compiled with `-O3`, and are the average of these twenty measurements. The confidence intervals (95%) are computed using Kalibera et al.'s formulation [59]. In addition to the traditional confidence interval, this formulation adds a factor accounting for the ratio between the actual and estimated (for lowest variance) repetition counts.

3.3.2 Performance Comparison

Figure 3.7 shows the speedup for each library or strategy for each of the platforms for the four PolyBench/C 4.2 benchmarks where the idioms are recognized and replaced. The outstanding performance of the BLAS libraries is due to a macro/micro-level design strategy. Such strategy enables higher performance gains with SYR2K than with other kernels as only half of the output matrix is accessed.

KernelFaRer never negatively affects performance in the case where no GEMM is detected because no transformation is imposed. Polly improves performance of other benchmarks in the PolyBench/C 4.2 suite that do not contain a GEMM idiom, but it occasionally causes a degradation. For instance, the performance for `atax` is only 70% performance of the baseline in the AMD environment.

Replacing a native GEMM idiom with a call to the BLAS library on the Intel platform leads up to a 2000 times speedup for the `2mm` benchmark. Polly also produces non-trivial performance gains — up to 200× for SYR2K on AMD. Polly only targets the first two levels of cache but the third-level cache is significantly larger (see Table 5.1) and its effective utilization is key for performance [35]. Polly also only applies thread-level parallelism to the outermost loop in a GEMM [18] while the libraries implement a dynamic run-time strategy to decide how to divide the *GEMMReduction* across multiple threads [94]. Polly's current GEMM-packing strategy could be improved by carefully choosing packing parameters for multithreading [81].

In the results shown in Figure 3.7, OpenBLAS [70] outperforms MKL, BLIS and ESSL, the vendor-specific solutions from Intel, AMD and IBM, respectively. This result contradicts the expectation that the performance of vendor-specific libraries should be on par with, or superior to, OpenBLAS. This is due to the largest dataset sizes in PolyBench/c 4.2 — $M = 2000$, $N = 2300$, and $K = 2600$ — not being in the optimal range for the vendor-specific libraries. Figure 3.8 shows the ratio of running times between a vendor-provided library (ESSL and MKL) and OpenBLAS. The results were measured for for matrix

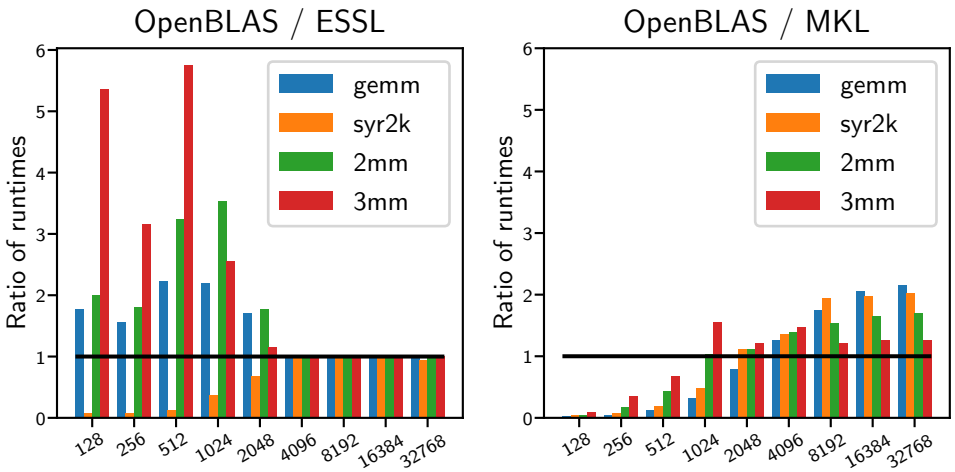


Figure 3.8: The ratio of runtimes of OpenBLAS and vendor libraries

sizes $M = K = N = 2^i, i \in [7, 15]$ and averaged over 10 runs. ESSL is up to 5× faster for `3mm` and on average 2× faster for GEMM and `2mm` than OpenBLAS for sizes smaller than 4K. The *cblas_syr2k* routine is better optimized in OpenBLAS than in ESSL, which explains the significantly poorer performance of ESSL. PPC's baseline binares (compiled with only `-O3`) also ran three times faster than Intel's baseline, which could explain its lower library speedups. MKL has been optimised for larger matrix operations, visible in its consistent improvment as dimension sizes increase. This is undoubtedly due to the insight that saving a percentage of time from an operation that takes several hours to complete is much more impactful than that same percent taken from one that takes a fraction of a second. Starting with the matrix sizes of 4K, all libraries show comparable performance for all kernels.

3.3.3 Robustness of Pattern Matching

Sophisticated programmers may attempt to improve the performance of a native-code implementation of GEMM by applying source-code level transformations. The BLISLab tutorial [47], made available by the Science of High-Performance Computing Group [86], provides a sequence of increasingly more sophisticated,

Table 3.4: Comparison of pattern matching tool robustness to different patterns. KernelFaRer is the presented method. Cells marked "X" indicate that the tool recognized and replaced the kernel idiom. "M" indicates instances where the tool only matched the kernel but was not able to replace it.

	gemm	syr2k	2mm	3mm	S1	S2	S3	S4
KernelFaRer	X	X	X	X	X	X	X	M
Polly	X		X	X			X	
IDL								

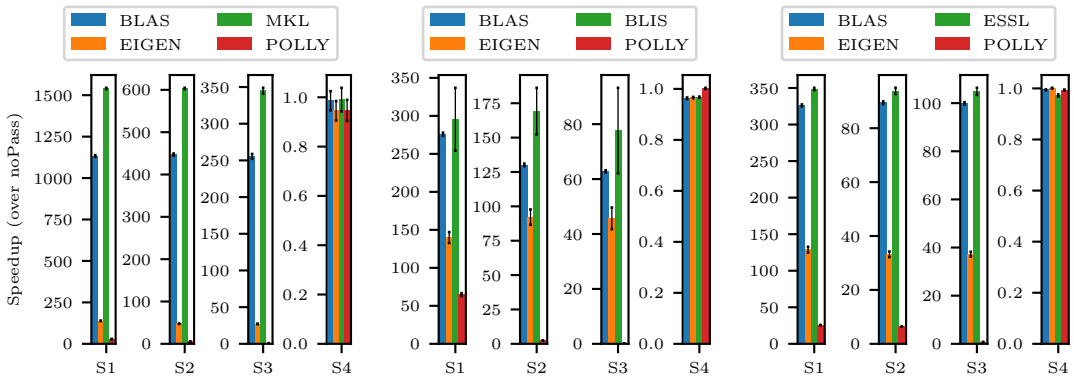
high-level-code-only implementations of GEMM[16]. `S0` is the simplest naïve form of GEMM where neither α nor β are used, all matrices are accessed in column-major order and can be expressed as $C = C + A * B$. In `S1` the matrix A is transposed. In `S2`, instead of transposing A, the loop nest (i, j, k) is interchanged to (j, k, i). `S3` both interchanges the loop nest, as in `S2`, and tiles the loop nest by 128, 64, and 128, respectively. `S4` interchanges the loop nest as in `S2`, tiles as in `S3`, and performs a gather operation to pack matrices A and B into blocks of tiles. The values used for tiling and packing are chosen to allow effective cache usage.

The more complex source code makes idiom recognition more challenging as shown in Table 3.4. KernelFaRer idiom recognition is robust in comparison with the two alternatives. Outside of the patterns in Polybench, the test suite used in its development, Polly recognizes only S3. IDL's recognition expects the result of the *GEMMReduction* to be stored into a scalar variable that is then later used to update the destination matrix — in all benchmarks the reduction is performed directly into the destination matrix. IDL is not expressive enough for the writing of an alternative specification to capture variants of GEMM. Capturing such variants would require modifications to the DSL compiler. This limitation highlights the brittleness of IDL's approach: a programmer must specify in a DSL the exact form of the code to be matched. A different specification is need for each subtle variation in the way the pattern is written. Operating in the LLVM IR, KernelFaRer avoids this gap between the

[16]Pattern matching assembly code is platform-specific and is out of the scope of this chapter.

Table 3.5: Comparison of time spent in LLVM passes implementing a GEMM & SYR2K detection method. Times are in milliseconds.

	gemm	syr2k	2mm	3mm	S1	S2	S3	S4
KernelFaRer	26.1	31.1	30.6	31.5	99.0	99.0	99.5	102.4
Polly	348.0	225.1	898.2	1400.5	390.0	616.3	2130.3	3268.5
IDL	255.0	-	959.6	2204.7	40.8	40.7	50.0	451.5



(a) Intel's x86 (b) AMD's x86 (c) IBM's PowerPC

Figure 3.9: The speedup relative to each hand optimization by replacing GEMM in each platform.

pattern description and the code to be matched because nodes in the pattern tree are explicitly made to match LLVM IR.

Figure 3.9 shows that even when a programmer spends significant effort to improve the source code of a GEMM kernel, highly specialized library implementations still deliver significantly superior performance. Note the different scales in the vertical axis of each of the graphs; the amount of work required to calculate the GEMM does not change, so the only varying factor is the effectiveness of the hand optimized version. For `S4` KernelFaRer identifies a `GEMM reduction` but the additional memory access needed to gather tiles of matrices A and B lead the KernelFaRer analysis to conclude that it is not safe to replace the GEMM idiom with library calls.

Polly improves the performance of the kernels `S1` and `S2` through loop tiling, unrolling and fusion. In the case of `S3` however, after applying GEMM specific optimizations, Polly produces a result matrix C that differs from that

computed by Eigen and BLAS for the same input matrices A, B, and C[17].

In addition to the programs listed in Table 3.4, we also created sixteen variants of the GEMM idiom by varying the presence of the scaling factors α and β, and the choice of first storing the result of the GEMM reduction into a scalar variable or storing it directly into matrix C[18]. KernelFaRer recognizes all 16 variants of GEMM; IDL only recognizes the four instances that accumulate into a scalar; and Polly does not recognize any of these variant as a GEMM. Polly always fails because all variants have an extra loop that stores the expression $C = \beta \times C$, or variants of it, generated by the compiler to guarantee correctness for cases where the innermost loop trip count is zero.

No False Positives Occurred

An important question is whether or not KernelFaRer ever detects a GEMM idiom where it does not exist. No such occurrences were detected while compiling the following benchmark suites: Rodinia 3.1 [17], SPEC CPU 2006 [43] and 2017 [13], and Nekbone[25]. None of the approaches found any instances of the GEMM idiom in these suites, though Nekbone already uses the CBLAS interface to perform GEMM operations. Both its design and the compilation of this extensive corpus of code provides confidence that KernelFaRer does not detect a GEMM idiom when none exists.

3.3.4 Flexibility

The matching code for GEMM is largely reused to match SYR2K pointing to the potential to extend this pattern matching approach to other kernels. The flexibility of this pattern-matching approach is underscored in this demonstration because SYR2K is different from GEMM in two principal ways: two matrix multiplication operations instead of one and a triangular output matrix iteration space. Matching SYR2K requires additional checks for the induction variables and their bounds and matrix layouts (similar to Table 3.1). Integrating the SYR2K pattern in KernelFaRer consists simply of adding a

[17] A communication to the authors resulted in no response up to the time of this writing.

[18] All the code used in this evalution is part of an Artifact submitted together with this manuscript.

match invocation to the *if* statement of the Algorithm 1. When extending the matching to other kernels, developer's time will be spent on filtering out false positive cases (e.g., rectangular iteration space) and matching corner cases, such as the addition of temporary variables by earlier compiler passes.

3.3.5 Effect on Compilation Times

Pattern matching consumes compilation time whether a program contains the pattern or not. Using LLVM's built-in pass-timing mechanism, this evaluation applies each of the idiom recognition methods to each benchmark and sums the wall-clock times for each of the passes added as part of the recognition method. Some analysis or transformation passes that are already performed for high levels of optimizations also support the recognition passes but these are not included in the summation because they are performed whether or not the idiom recognition is also performed. Table 3.5 shows the averages of 20 compilation-time measurements on PPC. Intel and AMD times differ by no more than 20% on average. The presented times include the time it takes to perform the corresponding pass (KernelFaRer, Polly or IDL) plus the time for `-O3` for KernelFaRer and Polly (because Polly requires `-O3`).

KernelFaRer has low impact in the compilation pipeline, scaling slowly as the complexity of the program increases. IDL's constraint solver method is inherently more costly than tree matching and must also analyse all function code, unlike KernelFaRer which knows that the idiom can only occur in loops of at least depth three and so limits its scope. The times from IDL are also solely for detection of the GEMM idiom because there is no replacement strategy. The higher compilation time of Polly includes more than idiom recognition as it also computes the polyhedral model of the loops, performs extra transformations beyond idiom recognition, and finally code generation back to IR. The additional compilation time in this evaluation are all from singular compilation units; these impacts will be magnified in larger programs or with more files. For instance, both PolyBench's gemm and `S1` are simple GEMM kernels; the only difference between the two is that `S1` transposes A. However, a verification loop elsewhere in the file that prints the result matrix

in `gemm` significantly increases the compilation time for Polly and IDL.

As Table 3.5 shows, the time KernelFaRer spends to detect and replace GEMM and SYR2K is about the same. This indicates that KernelFaRer's matching strategy runtime is not significantly affected by the complexity of the kernel. Polly does not have a specific matcher for SYR2K, but it is able to optimize SYR2K's loop nests in 225ms. The compilation time for IDL is not shown because, at the time of writing, there is no description of the SYR2K idiom in IDL.

3.4 Concluding Remarks

This chapter presents KernelFaRer, an idiom recognizer coupled with a data-flow analysis that finds and replaces idiomatic instances of a computing kernel with calls to high-performance libraries. KernelFaRer is implemented entirely within LLVM's compiler framework and extends its `PatternMatch` namespace to build a robust and effective tree-matching based pattern recognizer. The idiom recognition in KernelFaRer is robust: for GEMM it recognizes many more idioms than state-of-the-art approaches, e.g. Polly and IDL, including more complex instances of GEMM that were hand-optimized following well-known source-code transformations. No false-positives were produced when KernelFaRer analyzed extensive codebases from SPEC CPU 2016 & 2017, Rodinia, and Nekbone. This chapter also introduced a novel strategy to identify the access order of matrices in LLVM IR. Access-order information is central to correctly replacing kernels with calls to high-performance Basic Linear Algebra Subprograms (BLAS) libraries. KernelFaRer is a sophisticated idiomatic recognizer that only relies on standard data-flow analysis and extensions to tree-matching support in LLVM, a production-ready compiler framework.

Chapter 4

Fast Matrix Multiplication via Compiler-only Layered Data Reorganization and Intrinsic Lowering

New machine-learning algorithms, combined with the ever increasing demands of scientific and business analytics applications, highlight the importance of improving the performance of matrix algorithms and matrix multiplication in particular [88]. General Matrix Multiplication (GEMM) is a routine heavily used in high-performance computing and neural networks [90], both as a standalone operation and as a crucial component of other linear-algebra algorithms, such as LU decomposition. As the typical matrix sizes in GEMM operations for deep-learning workloads approach the order of 10,000 [79], efficiently partitioning the matrices to fit into the cache hierarchy is key to performance. The state-of-the-art approach in high-performance numerical libraries uses a layered approach for matrix multiplication that consists of (re)organizing the data to improve data locality as it moves from the main memory through the memory hierarchy and then relying on specialized assembly code to execute the multiplications efficiently in each targeted architecture.

This layered method, explained by Goto and Geijn [35] and used in both proprietary (e.g. Intel® MKL, IBM® ESSL) and open-source (e.g. OpenBLAS [93], BLIS [87], Eigen [38]) highly-optimized libraries, consists of a two-layer approach. First, a *macro kernel* performs tiling and packing of

the operand matrices across the caches. In a second layer, a *micro kernel*, implemented by means of compiler *builtins* or direct assembly instructions, extracts blocks from the packed tiles and executes the block multiplication. This work contributes new ideas to both the macro and micro kernels.

Libraries have been successful in exploiting the memory hierarchy to perform efficient matrix operations. However, they all share some drawbacks: 1. users must download and install architecture-specific libraries; 2. each library needs to be tailored by writing assembly code for every new architecture design; 3. manual changes to the users' code are necessary to call the libraries; and 4. often there is a time lag between the introduction of a new architecture and the creation of a specialized micro kernel for that architecture in a library. Overcoming these drawbacks would lead to broader utilization of the layered approach and of specialized micro kernels [90].

Linear-algebra libraries share similar code for matrix multiplication because they all make use of the ideas described by Goto and Geijn [35]. This insight leads to the first contribution of this chapter, which aims at capturing the layered strategy described by Goto and Geijn in a compiler-only LLVM-based optimization pass. Implementing the layered strategy as a general-purpose compiler pass brings three benefits. 1. programs written in all languages that are supported by the LLVM frontend (e.g., C, Fortran, Go, Rust) can leverage the strategy when there is no library interface or a library implementation is outdated; 2. the tiling and packing code, which is common to all libraries, is automatically generated by the new compiler pass; 3. the algorithm to lower the LLVM intrinsic to architecture-specific micro-kernel code only needs to be implemented once for each architecture.

The resurgence of machine learning also led to the introduction of application-specific accelerators into general-purpose CPUs. Examples include Intel®'s Advanced Matrix Extension (AMX) [53], Arm's Matrix Extension (ME) [5] and IBM POWER10's Matrix Multiply Assist (MMA) [78], [84]. AMX is an off-core accelerator with a dedicated register file that employs inner product operations to compute in-register matrix multiply using novel tile registers [53]. New instructions allow the CPU to communicate with AMX through an accelerator

command queue [53]. Both Arm's ME and IBM's MMA unit are on-core extensions that use SIMD vector registers for input and output operands. However, ME and AMX rely on inner products for multiplication, while MMA uses outer products.

This heterogeneity of accelerators further complicates the hand optimization of micro kernels that library developers must perform. In a compiler-only code generation path, the adoption of LLVM's intermediate representation (IR) `llvm.matrix.multiply` intrinsic abstracts target-specific operations under a clear interface reducing the need for specialized micro kernels to a single implementation of the intrinsic. This LLVM intrinsic computes the product of two fixed-size matrices. LLVM provides a generic lowering algorithm that unrolls the matrix-multiply computations to target-independent IR code. LLVM's backends can then further lower IR code to target-specific machine code for any of the many backends supported. For instance, another contribution of this chapter is a lowering algorithm of the LLVM `llvm.matrix.multiply` intrinsic that efficiently utilizes the new MMA unit in POWER10. When generating code for MMA this intrinsic computes a fast outer-product-based matrix multiplication for the micro kernel.

Summarizing, this chapter makes the following contributions:

- An algorithm for a compiler-only, architecture-independent, tiling & packing strategy for the macro kernel that improves upon the strategy described by Goto et al. [35] (Section 4.1.1). The incorporation of this algorithm in LLVM leverages all available backends for processor architectures by building upon a compiler-intrinsic micro kernel instead of a hand-crafted assembly micro kernel as in most high-performance BLAS libraries;

- An algorithm to lower the LLVM `llvm.matrix.multiply` intrinsic (micro kernel) to the new IBM MMA extension (Section 4.1.2). The specialized code generated for MMA benefits not only the code generated by the macro-level algorithm, but any compilation path that uses the intrinsic.

- A thorough experimental evaluation that shows that: 1. the proposed

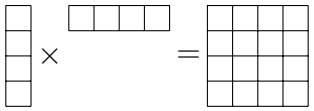


Figure 4.1: Outer-product (rank-1 update) operation.

macro-level algorithm, even when coupled with a generic intrinsic lowering, can perform more than 22× faster – for small matrices on Intel®– and more than 6× faster – for large matrices on POWER9™– than PLuTO[11], a widely used polyhedral optimizer. 2. this compiler-only approach generates code that is on-par with `Eigen` and 34% slower than BLAS on Power9™; 3. coupling the macro-level algorithm with an MMA-specific lowering of the `llvm.matrix.multiply` intrinsic in POWER10 achieves: (a) more than 2.6× the performance of the VSX micro kernel; (b) 10% more performance than `BLAS` for a small SGEMM and up to 96% of `BLAS` peak performance for large SGEMM; (c) 83% faster code than `Eigen` for large matrices.

An overview of the POWER10 Matrix-Multiply Assist (MMA) facility follows in Section **??**. Then, Section 4.1 describes the proposed code generation approach, detailing the algorithms for the macro and micro kernels. Section 4.2 presents and analyzes the experimental results. Section 6.2 describes and puts in perspective the works related to this chapter, and finally, Section 4.4 presents conclusions.

4.1 Code Generation for GEMM

Algorithm 3 provides an overview of the computation of GEMM within our compiler pass. The code generation can be divided into two abstract levels: **macro:** target-independent blocking and packing of matrices for faster memory access and **micro:** small target-dependent kernel lowered from compiler-intrinsic calls. A compiler-intrinsic-based micro kernel, instead of a hand-crafted assembly micro kernel, enables the micro-kernel code to be (i) automatically optimized — for instance via vectorization — and mapped — via instruction selection — to efficient instructions in the target or (ii) generated by a lowering algo-

Algorithm 3 Algorithm overview for GEMM

```
1: for j ← 0, N, step nc do
2:   for k ← 0, K, step kc do
3:     pack(B, BPack, k, j, kc, nc, kr, nr, "B", "Row")
4:     for i ← 0, M, step mc do
5:       pack(A, APack, i, k, mc, kc, mr, kr, "A", "Col")
6:       for jj ← 0, nc step nr do
7:         for ii ← 0, mc, step mr do
8:           AccTile ← 0
9:           for kk ← 0, kc, step kr do
10:            BTile ← loadTile(BPack, kk, jj, kr, nr, ldb)
11:            ATile ← loadTile(APack, ii, kk, mr, kr, lda)
12:            ABTile ← llvm.matrix.multiply(ATile, BTile, mr, kr, nr)
13:            AccTile ← ABTile + AccTile
14:          CTile ← loadTile(C, i + ii, j + jj, mr, nr, ldc)
15:          if k == 0 then
16:            CTile ← β×CTile
17:          CNewTile ← α×AccTile
18:          CTile ← CTile + NewCTile
19:          storeTile(CTile, C, i + ii, j + jj, mr, nr, ldc)
```

rithm that aims to exploit target-specific instructions (See Section 4.1.2). The macro-level strategy is inspired by the memory-hierarchy modelling described by Goto and Geijn [35]. Nevertheless, our compiler-only macro-level algorithm differs from Goto and Geijn's seminal work on key aspects that better capture modern CPU's and accelerator's features (See Section 4.1.1). Implementing this strategy fully inside the compiler has advantages: 1. compile-time known features of the target architecture — e.g. minimum vector-register length — guide the automatic generation of tiling & packing code, contrasting with hand-crafted and target-specific implementations in BLAS libraries; 2. a flexible packing layout enables the use of either row or column-major tiles to match the access order in the micro kernel; 3. defining tile sizes for more levels or for different cache/memory organizations only requires changes in the heuristic code itself — the remaining algorithm code remains untouched; and 4. the packing code — generated by the compiler instead of hardcoded as in libraries — can be retargeted to accelerators with, for instance, explicit cache/memory management (e.g. software-managed caches or scratch-pad memories).

In Algorithm 3, `j`, `k`, and `i` are the offsets of the blocks in the matrices, counted in terms of elements. The values `lda`, `ldb` and `ldc` are the leading dimensions, and thus the access stride, of the matrices as they are originally stored in memory. The blocking parameters `mc`, `kc`, and `nc` divide the input matrices `A` and `B` into blocks properly sized for cache. The `pack` functions in lines 3 and 5 each create a buffer in main memory containing one copy of an entire block of matrix `B` or of matrix `A`. The last argument of the function `pack` specifies the data layout within each tile. The storage order of the tiles after packing, which is independent of the original storage order of the elements of `A` and `B`, is selected to benefit the tiled multiplication (see Section 4.1.1).

The tiling parameters `mr`, `kr`, and `nr` ensure optimal resource utilization in the micro kernel. `ATile`, `BTile`, `AccTile`, `ABTile`, `CTile`, and `CNewTile`, are virtual LLVM IR vectors. These vectors are allocated and loaded to physical registers by subsequent code-generation passes. Lines 10 and 11 each load a single tile of an input matrix into an LLVM IR vector from the packed buffers. The algorithm invokes the intrinsic on Line 12 which multiples `ATile` and `BTile` and results in a new tile (`ABTile`) that is accumulated into `AccTile`. `AccTile` is kept in vector regiters for all iterations of the loop on Line 9. On the MMA code generation path, `AccTile` is mapped to accumulator registers. A tile of matrix `C` is loaded into `CTile`, in line 14, and scaled by the constant β on the first iteration of the loop on Line 2, satisfying GEMM's requirement that the matrix multiplication updates the values of the destination matrix. Likewise, the accumulation produced in `AccTile` is multiplied by the constant α in Line 17 satisfy the GEMM mathematical expression. Once the entire multiplication is completed, `CTile` can be stored to its position in memory (Line 19).

4.1.1 Macro-level Algorithm: blocking, tiling and packing

Assuming large matrices, portions of each matrix must be brought, through the memory hierarchy, to the registers. These portions become operands to a mulitiplication intrinsic. At the micro level, a code generation pass lowers

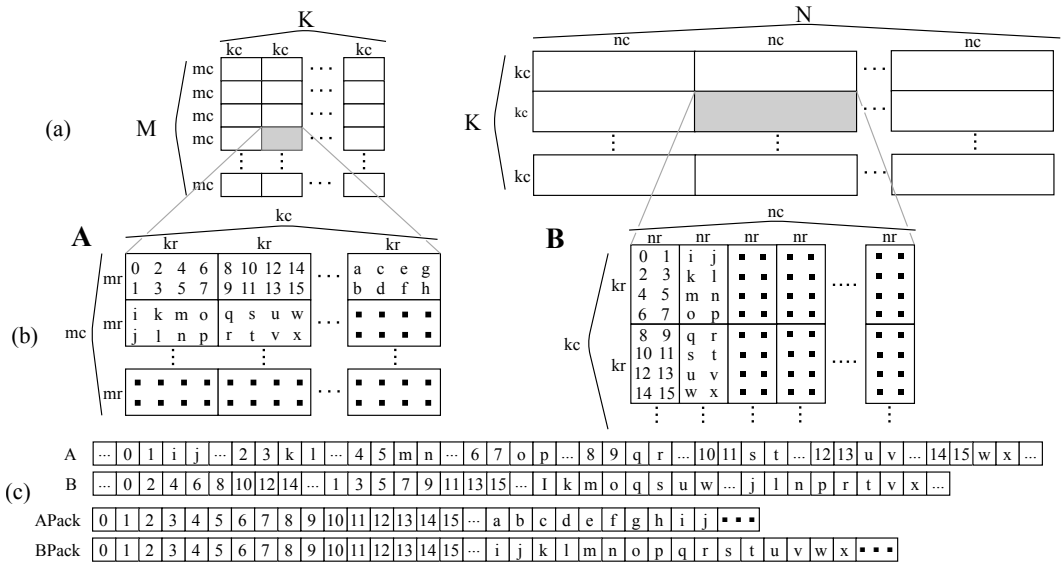


Figure 4.2: Tiling and packing for `llvm.matrix.multiply`.

the intrinsic to specific hardware instructions (Section 4.1.2). Contrary to other approaches, no manual vectorization or hand-written assembly code is required. The whole micro kernel development happens inside the compiler as the intrinsic takes advantage of the target's available matrix engine operations.

In the example shown in Figure 4.2 (a), matrix `A` has $M \times K$ elements and matrix `B` has $K \times M$ elements. Both matrices are stored in memory in column-major order. A block of matrix `A` has `mc` $\times$ `kc` elements and a block of matrix `B` has `kc` $\times$ `nc` elements. Each block of `A` is divided into tiles of `mr` $\times$ `kr` elements and each block of `B` is divided into `kr` $\times$ `nr` tiles. Figure 4.2 (b) presents the order of the elements and tiles in the packed block when `mr` $= 2$, `kr` $= 4$ and `nr` $= 2$. These small values are used to keep the figure at a reasonable size. The actual values for `mr`, `kr`, and `nr` are selected at compilation time to result in a performant micro-level computation for each data-type size in each architecture.

Figure 4.2 (b) illustrates the partition of a block into tiles. The tiles are packed within the block in the order in which they will be accessed in the innermost tiling loop. That is, disregarding the preferred layout of elements within each tile for a particular architecture, the tiles will be placed in rows in the block `mc` $\times$ `kc` of `A` and in columns in the block `kc` $\times$ `nc` of `B`. The

sequential numbers inside the tiles are used simply to indicate the order in which the elements will be accessed when micro-level tiles are loaded into vector registers. Letters are used at the end of the first row (column) and at the start of the second row (column) of tiles to indicate the relative order of these elements. Figure 4.2 (c) shows the layout of the elements of each matrix as they are originally stored in column-major order in memory, and the order of the elements of `A` and `B` as stored into the `APack` and `BPack` buffers after packing. The layout of elements within the tiles is tailored to the needs of the underlying architecture. For example, Arm ME, expects a row-major `A` and a column-major `B` for its input operands and accumulates the result in a row-major C [5]. Figure 4.2 (c) presents an example for IBM® POWER10 MMA, which uses column (`A`), row (`B`), row (`C`) layouts.

$$\texttt{kc} \leq \mathit{L1SizeInBytes}/2/\mathit{TypeSizeInBytes}/\texttt{VL} \tag{4.1}$$

$$\texttt{kl} \leq (\mathit{L1SizeInBytes}/2/\mathit{TypeSizeInBytes} - \texttt{VL} \times \texttt{VL})/(2 \times \texttt{VL}) \tag{4.2}$$

$$\texttt{mc} \leq (\mathit{L2SizeInBytes} - \mathit{L1SizeInBytes})/\mathit{TypeSizeInBytes}/\texttt{kl} \tag{4.3}$$

$$\texttt{nc} \leq (\mathit{L3SizeInBytes} - \mathit{L2SizeInBytes})/\mathit{TypeSizeInBytes}/\texttt{kl} \tag{4.4}$$

$$\texttt{kc}\ mod_2 \texttt{kr} = 0 \tag{4.5}$$

$$\texttt{mc}\ mod_2 \texttt{mr} = 0 \tag{4.6}$$

$$\texttt{nc} mod_2 \texttt{nr} = 0 \tag{4.7}$$

Constraints 4.1-4.7 model how the tiling & packing factors are computed by the compiler-only macro-level algorithm. `LXSizeInBytes` is the number of bytes in cache level `X`, `TypeSizeInBytes` is the number of bytes in the matrix operations (e.g 4 for single-precision floating-point numbers), and `VL` is the minimum vector length size on the target architecture (e.g 4 for 128-bit vector registers). Similar to Goto & Van Geijn [35], the macro-level algorithm allocates a significant portion of L1 for a piece of each `kc` × `nc` block of `B`. However, different from Goto & Van Geijn, which allocate half of L1 for a `kc` × `nr` piece, the macro-level algorithm allocates half of L1 only for a `kc` × `VL` piece of `B`'s `kc` × `nc` block (Constraint 4.1). This strategy produces a larger value for `kc` to

exploit the fact that most modern architecture have enough vector registers to hold `nr` multiples of `VL`. The algorithm considers that the remaining half of the L1 will be used to hold `VL` $\times$ `VL` C elements from memory and `VL` elements of `A` and `B` as per Constraint 4.2. `kl` is used to maximize the use of L2 for a `mc` $\times$ `kl` piece of an `mc` $\times$ `kc` block of `A` and to determine how much to allocate in L3 for the `kl` $\times$ `nc` piece of `kc` $\times$ `nc` block of `B`. Constaints 4.3 and 4.4 consider the effective size of L2 and L3, respectively, and take into account cache inclusion as in most modern architectures. The inclusion property of caches is not explicitly modelled by Goto & Van Geijn. A final constraint is to make `kc`, `mc`, and `nc` multiple of their respective register-tiling factor in the micro kernel, `kr`, `mr`, `nr` (Constraints 4.5-4.7). POWER10's MMA features eight 512-bit accumulators and thirty two 128-bit vector registers to support outer-product computations. Therefore, for 32-bit matrix elements the performant choice is `mr` = 8 and `nr` = 16, as explained in Section 4.1.2, while `kr` is selected to maximize the number of in-accumulator operations. The effective sizes for L2 and L3 caches may depend on the machine load. For instance, L3 caches can be shared on some configurations of POWER10 (Section 4.2), and this allows a single-threaded computation to use all of the core's caches. To account for this effect, the macro algorithm has command line options to provide the effective L2 and L3 cache sizes available per core.

Parameters `mr`, `kr`, and `nr` define the size of the GEMM computed by the micro kernel at each innermost loop iteration in Algorithm 3. The values of these parameters are selected by the intrinsic implementation, based on the architectural design of the target. Section 4.1.2 provides an explanation of how optimal `mr`, `kr`, and `nr` need to be chosen for the micro kernel on POWER10 MMA. The values of `mr`, `kr`, and `nr` are the only parameters that need tuning as the cache blocking parameters `mc`, `kc` and `nc` are chosen based on target-specific cache size information available in LLVM. Following the blocking strategy described by Goto and Geijn, the value of `kc` is selected in such a way that an entire row of tiles of the matrix `A` — `mr` $\times$ `kc` elements — and an entire column of tiles of the matrix `B` — `kc` $\times$ `nr` elements — fit simultaneously in the L1 data cache. The L1 cache must also have space for an

`mr` $\times$ `nr` tile of the matrix `C` but such a tile is quite small in comparison with the space needed for tiles of `A` and `B`.

Figure 4.2 (b) shows the order of the matrix elements in the buffers after packing, while Figure 4.2 (a) show the original matrices. Arrays `A` and `B` in Figure 4.2 (c) reflect that the original matrices are stored in column-major layout. Buffers `APack` and `BPack` in Figure 4.2 (c) illustrate that, after packing, the elements of matrices `A` and `B` are stored into these buffers in the order in which they will be accessed by the micro-level computation. Packing provides two benefits. First, the tiles of matrices `A` and `B` lie in the copied memory in the order they will be loaded for the multiply intrinsic in lines 10 and 11. Second, each row of tiles of `A` residing in a part of L1 cache is used in $\lceil \frac{nc}{nr} \times \frac{kc}{kr} \rceil$ multiplications ($\lceil \frac{mc}{mr} \times \frac{kc}{kr} \rceil$ times for a column of tiles of `B`).

When `CNewTile` is not a multiple of a micro tile, the remainder elements are filled with zeroes in the packing buffers and the result is stored with scalar store instructions instead of the efficient matrix store intrinsic used for full tiles. This brings an overhead for using slow, element-by-element stores instead of fast strided stores. For the remaining and zero-padded elements, the micro kernel still performs a full computation.

4.1.2 Micro-Level Algorithm

The micro-level algorithm is centered around the LLVM intrinsic `llvm.matrix.multiply`. Intrinsics encapsulate a computational idiom and enable specialization to specific architectures, thus preventing the duplication of code within a compiler infrastructure. An intrinsic can be viewed as a function call that is eventually replaced by inlined generated code. Like a function call, an intrinsic has input parameters and a return value. An intrinsic can be transparently lowered to target-agnostic or to target-specific code. As expected, the target-agnostic code has lower performance but is available for all LLVM-supported backends. This section describes the design process to create a target-specific lowering pass for the `llvm.matrix.multiply` intrinsic that utilizes the MMA instructions to deliver high performance matrix multiplication.

The LLVM IR fragment in Listing 4.1 shows how the `llvm.matrix.multiply`

Listing 4.1: An example usage of `llvm.matrix.multiply`.

```
%CNewTile = call <128 x float>
  @llvm.matrix.multiply.v128f32.v40f32.
  v80f32(%ATile, %BTile, 8, 5, 16)
```

intrinsic — called in line 12 of Algorithm 3 — takes `ATile` and `BTile` to produce `CNewTile`. The mangled function name shows the types of `%ATile`, `%BTile`, and `%CNewTile`. The return value, `%CNewTile` is `<128 x float>`, an 8×16 result, while `%ATile` is `<40 x float>`, an 8×5 tile, and `%BTile` is `<80 x float>`, a 5×16 tile. The three remaining parameters represent the dimensions of the matrices, `mr` $= 8$, `kr` $= 5$, and `nr` $= 16$, indicating that this intrinsic computes a $C_{8\times16} = A_{8\times5} \cdot B_{5\times16}$ multiplication. These tile sizes must be known at compilation time and must match the dimensions of the packed input and output vectors. The requirement that the tile sizes are known at compile time allows both the target-agnostic lowering and the MMA target-specific lowering to completely unroll the matrix-multiplication loop nest in the IR to give the backend full control over instruction rescheduling.

In addition to these software constraints, scheduling decisions in the algorithm that lowers the intrinsic computation to execute in an MMA backend must also adhere to the following hardware constraints: 1 there are at most eight accumulators available per thread and for each accumulator that is used, the usage of four VSRs are blocked; 2 there are 64 VSRs, thus if eight accumulators are used, there are 32 VSRs remaining to contain the data from the input matrices; 3 two multiply-and-accumulate outer-product instructions can be issued on a single cycle; 4 the issue-to-issue latency for the same accumulator is four cycles; and 5 spilling an accumulator to memory is an expensive operation because it requires an instruction to disassemble the accumulator into four VSRs, four vector store instructions and, later, four vector load instructions.

Figure 4.3 illustrates how `CNewTile` is divided into portions that are assigned to the MMA accumulators. `ATile`, `BTile`, and `CNewTile` are represented in two dimensions to illustrate the position of the elements in the matrices. Each small square in the figure represents one 32-bit element of a matrix. A circled

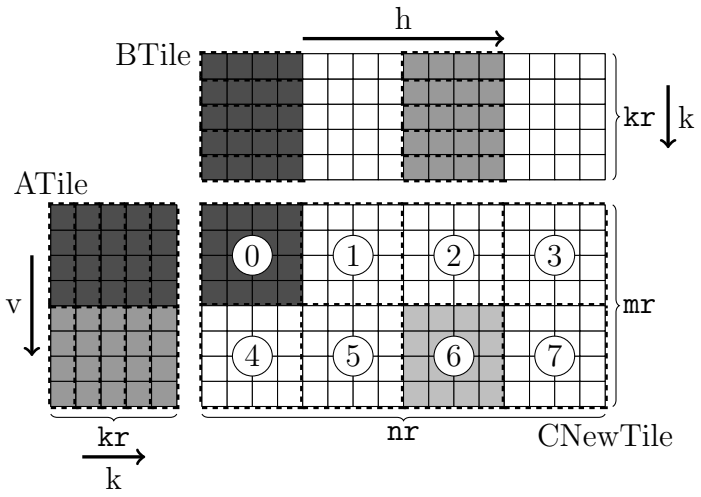


Figure 4.3: Division of `CNewTile` into MMA accumulators.

number indicates that the corresponding portion of `CNewTile` is assigned to that accumulator number. When the intrinsic is executed each accumulator computes `kr` outer products using a multiply-and-add operation. The two tones of gray colour in Figure 4.3 illustrate that a strip of `ATile` and a strip of `BTile` are used for the accumulation of each portion of `CNewTile`. Each strip is reused for all the accumulations in the same row or column of accumulators. Each outer-product computation needs two four-element operands, one from `ATile` and one from `BTile`. These operands are surrounded by dashed lines for the two accumulations highlighted in gray. The arrows indicate how the loop indices in Algorithm 4 iterate for the example in the figure.

Algorithm 4 describes the lowering of the intrinsic computation for MMA. The compile-time constants `VAccs` and `HAccs` (used in lines 5 and 7) specify the layout of the accumulators for the computation. For the example in Figure 4.3, `VAccs = 2` and `HAccs = 4`. These constants in the compiler generalize the lowering and make it applicable to future architectures where the ideal arrangement to increase data reuse may be different from the 2×4 arrangement in the POWER10 processor.

After creating `CNewTile` (line 2) and assembling and zeroing all the accumulators (line 3), Algorithm 4 iterates `k` from 0 to `kr` $-$ 1 (line 4) to extract operands from `ATile` and `BTile` into virtual IR registers. For each value of k, using the accumulator assignment shown in Figure 4.3, the algorithm extracts

Algorithm 4 Computation by `llvm.matrix.multiply`

```
1:  function LLVM.MATRIX.MULTIPLY(ATile, BTile, nr, kr, mr)
2:    CNewTile ← Empty
3:    Assemble ACCs and initialize to zero
4:    for k = 0 to kr-1 do
5:      for v = 0 to VAccs-1 do
6:        AOps[v][k] ← Extract op from ATile[v][k]
7:      for h = 0 to HAccs-1 do
8:        BOps[h][k] ← Extract op from BTile[h][k]
9:    for k = 0 to kr-1 do
10:     for v = 0 to VAccs-1 do
11:       for h = 0 to HAccs-1 do
12:         Accs[v][h] +=
              MMABuiltIn(AOps[v][k], BOps[h][k])
13:   Disassemble ACCs and store VSRs into CNewTile
14:   return CNewTile
```

two operands from `ATile` (lines 5-6) and four operands from `BTile` (lines 7-8). For $k = 0$ the operands are extracted from the leftmost column of `ATile` and from the top row of `BTile` in Figure 4.3. The algorithmic presentation in Algorithm 4 uses the notation `AOps[v][k]` and `BOps[h][k]` to show the connection between the operands extracted from `ATile` and `BTile` with the use of these operands in the `MMABuiltin` on line 12. In the compiler, at this point in the lowering, each four-element operand is extracted into a virtual IR register. The actual VSRs used for each operand are determined later by a register-allocation pass.

In Figure 4.3 each operand is formed by four elements and, once extracted, occupies one 128-bit VSR. Given constraints 1 and 2, with the choice of `kr` = 5, there are enough non-blocked VSRs to contain all the thirty operands needed for the computation illustrated in Figure 4.3. Thus, laying out the accumulators in this 2×4 pattern maximizes the reuse of values loaded into the VSRs: operands extracted from `ATile` are reused four times and operands extracted from `BTile` are reused two times.

Once all operands are extracted into VSRs, the algorithm again iterates over the dimension `kr` to compute each piece of `C` to avoid spilling any accumulator to memory. Following constraint 3, two outer-product instructions are issued

in each cycle. Four pairs of accumulators can be scheduled before circling back to the first pair, thus satisfying constraint 4. The assignment of a portion of `CNewTile` to a single accumulator eliminates the need to spill accumulators, thus increasing the performance according to constraint 5.

The lowering of the intrinsic for execution in POWER10 is based on a set of builtins that encapsulate the computation of an outer product. There is a set of builtins for each data type to allow the code generator to select a multiply-and-add that either initializes or updates an accumulator. All combinations of positive/negative multiplication with positive/negative accumulation are available as well. For some data types there are also builtins that perform saturating arithmetic instead of overflow for accumulation. Thus, when lowering the intrinsic for the GEMM computation, the compiler selects the appropriate positive multiply and positive accumulate builtin for the specified data type which is then used on line 12.

4.1.3 Other Data Types

The presentation so far assumed 32-bit data types where each operand VSR contains 4 elements and an MMA instruction computes a rank 1 update, computing and accumulating a single outer product. Halving the data-type size doubles the number of elements in each VSR and doubles the rank of the update. For example, for a 16-bit data type MMA computes a rank 2 update while an 8-bit data type computes a rank 4 update. The packing of more elements into a single VSR and the accumulation of multiple outer products by a single MMA instruction requires changes to Algorithm 3 and Algorithm 4. Let n be the number of outer products performed by an MMA instruction — i.e. the rank of the update. Now the step size of the loops on lines 4 and 9 must both be n because, in Figure 4.3, n rows of `BTile` and n columns of `ATile` are packed into each VSR. The extraction of operands in lines 6 and 8 is now a strided access. For instance, for $n = 2$ (16-bit data types), four consecutive elements are extracted from row k and four consecutive elements are extracted from row $k + 1$ to form the 128-bit VSR. The length of `kr` must increase by n times to provide enough data to populate the VSRs. The effect is that more

partial-product accumulations can be computed per micro-kernel invocation given the same number of assemblies and disassemblies because the number of multiplications per outer product increases by n.

For double-precision floating-point data type `f64` an accumulator contains 4×2 64-bit elements. The operand extracted from `ATile` is placed into a combination of two VSRs that together contain four elements while the operand extracted from `BTile` is placed into a single 128-bit VSR containing two elements. Therefore, for `f64` the value of `nr` should be reduced in half to reflect the number of VSRs available. With this reduction, an `ATile` tile occupies 16 VSRs and a `BTile` tile also occupies 16 VSRs. The extraction of operands into vector registers in lines 6 and 8 of Algorithm 4 must be changed accordingly.

4.1.4 Arbitrary Values for `nr`, `mr`, `kr` and Access Order

Until now, the algorithms have used values of `mr` and `nr` selected such that a micro kernel with the accumulator arrangement shown in Figure 4.3 could be computed with a single set of assemble and disassemble instructions. However, the implementation of Algorithm 4 in LLVM must handle any `llvm.matrix.multiply` intrinsic created by any compilation path and thus must handle arbitrary values for `nr`, `mr` and `kr`. The code-lowering algorithm also supports inputs and outputs in any access order through modifications to the functions that extract operands and store the results in the accumulators to memory.

To handle larger values of `mr` and `nr`, the micro-level code-lowering algorithm has an additional outer double-nested loop that logically divides the `CVec` tile into 8×16-element sections as shown in Figure 4.3. Each of these sections can then be handled as shown in Algorithm 4. The disadvantage of a tile size that spans multiple accumulator sections is that the extraction of data into vector registers becomes more complex. For example, consider a 32-bit data multiplication as shown Figure 4.3 but with the values of `nr` and `mr` double of what is shown in the figure. The rows of `ATile` and `BTile` shown in Figure 4.3 are now a portion of the rows of larger tiles and the data extraction must gather the correct data into the vector registers that will be used by the accumulators.

This data gathering adds additional code and may impact access locality if the tiles are large enough. Moreover, if Algorithm 4 is used in combination with Algorithm 3, then the packing work done earlier in lines 3 and 5 of that algorithm may not result in optimal locality. As well, if `kr` is smaller than K (see Figure 4.2) multiple invocations of the intrinsic are needed to compute each element of the result matrix. Therefore, accumulators must be assembled and disassembled multiple times, creating an issue with constraint 5.

4.2 Experimental Evaluation

Experimental results support the following claims: 1. the macro-level algorithm is architecture-independent: it is performant across four different architectures (Intel®'s and AMD®'s x86; IBM®'s Power9™ and POWER10); 2. the algorithm, when fully implemented inside the compiler, surpasses the performance of PluTo, a widely used polyhedral-based compiler-only approach; 3. the macro-level algorithm, even when coupled with a generic-lowering of the LLVM's matrix-multiply intrinsic, approaches the performance of Eigen [38] and OpenBLAS [93]; 4. for small GEMMs the compiler-only approach performance can surpass library calls; 5. the MMA lowering delivers more than 2.6x the performance of VSX on POWER10; and 6. our compiler-only solution boosts GEMM's performance by exploiting target-specific matrix engines: the micro-level algorithm lowers multiply-add GEMM reductions to efficient IBM's MMA instructions resulting in better performance than Eigen and up to 96% of OpenBLAS' peak performance.

Machine Setup

Experimental evaluation used the four platforms shown in Table 5.1[1]. All platforms run Linux with 64-bit kernel at version `4.15.0-155-generic`, except POWER10 which runs at version `4.18.0-277.el8.ppc64le`. L1 and L2 cache sizes are per core while L3 is shared among all cores on Intel®'s and AMD®'s

[1]Core counts are per socket and thread counts are per core. Cache sizes in POWER10 are those available for a single thread.

Table 4.1: Machine configuration used in the evaluation.

Component / Processor	Intel® Xeon™ 8268	IBM® Power9™	IBM® POWER10	AMD® EPYC™ 7742
Cores/Threads	24/2	20/4	8/8	64/2
L1 i-cache	32KiB	32KiB	32KiB	32KiB
L1 data cache	32KiB	32KiB	48KiB	32KiB
L2 (unified)	256KiB	512KiB	1024KiB	512KiB
L3 (unified)	35.75MiB	10MiB	4MiB	16MiB
RAM	755GiB	1TiB	1TiB	995GiB
Memory bandwidth	131.13GiB/s	140GiB/s	–	190.7GiB/s

machines. L3 cache is shared between pairs of cores in Power9™ but is local to a core in POWER10.

Compiler Options

All binaries are compiled with Clang version 14 at `-O3` targeting each architecture. Binaries for Intel® are targeted and tuned to Cascade Lake™ (`-march=cascadelake -mtune=cascadelake`), AMD®'s to Zen 2™ (`-march=znver2 -mtune=znver2`), and IBM®'s machines to their corresponding CPUs (`-mcpu={power9|power10} -mtune={power9|power10}`). All binaries are statically linked (`-static`).

Code Generation and Libraries

Section 4.2.2 and Section 4.2.3 show results for the following code generation strategies:

- `Intrinsic`: GEMM loops are replaced with a single call to LLVM's matrix-multiply intrinsic. This option causes all GEMM loops to be unrolled and multiply-add computations to be lowered with either generic or MMA lowering (Section 4.1.2).
- `Tiling`: This option tiles the GEMM loops iterating over each dimension (M, K, and N) and calls the matrix-multiply intrinsic in the body of the innermost loop (depth 3). Tile sizes are computed following Goto et al.'s strategy as described in Section 4.1.1.

- `Tiling+Packing`: This option tiles the GEMM loops and packs the input matrices `A` and `B` as described in Section 4.1.1. The matrix-multiply intrinsic is called in the body of the innermost loop (depth 6) to compute a block of `C` from tiles of `A` and `B`.

- `PLuTo`: This option applies a source-to-source transformation using PLuTo[2], a polyhedral-based paralelism and locality optimizer, that automatically tiles and redorders loops annotated with `#pragma scope`. PluTo's auto-tiling[3] for both the first and second-level caches are enabled.

- `BLAS`: This option replaces the GEMM loops with a single call to the CBLAS interface GEMM routine. The OpenBLAS[4] [93] library is the implementation of the CBLAS interface evaluated. The library is compiled and linked as described in Section 4.2.

- `Eigen`: This option replaces the GEMM loops with Eigen[5] [38] code to compute the general matrix-matrix multiply. Eigen code is compiled as described in Section 4.2.

The experiments use versions of `BLAS` and `Eigen`, which support both VSX and MMA instructions, that were contributed to the open source community by internal teams from IBM®. All versions listed above are compiled and set to execute single-threaded code. A naive implementation of a $M \times K \times N$ SGEMM in C++ serves as the base code for our algorithm. Carvalho et al.'s pattern-matching algorithm [16] is used to automatically identify and replace these GEMM loops in the source code with the code generated by the macro-level algorithm. The input matrices are stored and accessed in column-major order and the values $\texttt{mr} = 16$, $\texttt{nr} = 4$, and $\texttt{kr} = 64$ are used for all platforms, except on POWER10 which used $\texttt{mr} = 16$, $\texttt{nr} = 8$, and $\texttt{kr} = 128$.

Even though `PLuTo` lacks critical optimizations, such as packing, it is the only compiler-only solution available that produced correct results for

[2]PLuTo Release v0.11.4
[3]`-tile -l2tile`
[4]OpenBLAS `1e4b2e98d953a18df85243a3fa019a105cbcb3dc`.
[5]Eigen `5bbc9cea93ef29cee2b8ffb2084d4ebca32600ba`

the programs used in the experimental evaluation. Polly, another compiler-only solution, either failed to optimize the input code or generated code that computes incorrect results.[6]

Experimental Methodology

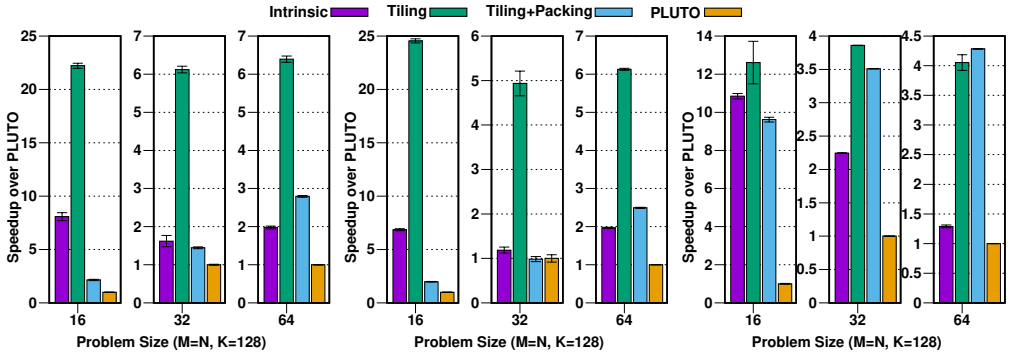
The methodology follows these steps: 1. Compile each benchmark, for each platform, with aggressive optimization flags (see Section 4.2) using the six strategies: `Intrinsic`, `Tiling`, `Tiling+Packing`, `PLuTo`, `BLAS`, and `Eigen`. 2. Create a list containing all of the executables. 3. Measure the execution time of each element of the list once. 4. Randomize the list of executables before the next set of measurements. 5. Repeat until there are twenty measurements for each executable. This methodology ensures that changes to the execution environment that may affect performance manifest as a higher variance between executions of the same executable rather than as a bias in the measurements for a version of the experiements. Section 4.2.2 and Section 4.2.3 show 95% confidence intervals.

4.2.1 Performance Comparison Against Other Compiler-Only Approaches

The results in Figure 4.4 indicate that for small SGEMM sizes the performance of `Tiling` is far superior than `PLuTo` across all platforms: `Tiling` is up to 22x faster than `PLuTo` for the smallest SGEMM size on Intel®, almost 25x faster on AMD®, and over 11x faster on Power9™. For small problem sizes `Tiling`

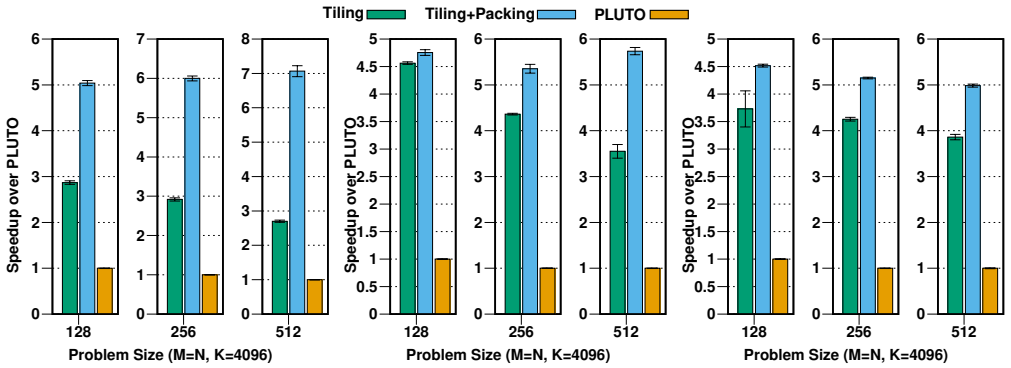
[6]These incorrect results were reported to the developers of Polly.

This chapter presents an end-to-end solution that compiles a C/C++ input program and replaces identified GEMM loops with the high-performance code generated by the macro-level algorithm. Another compiler-only solution, introduced by Uday Bondhugula, is not an end-to-end solution because it relies on a hand-crafted MLIR code written in the Affine dialect [10]. At the time of writing, there is no way to automatically translate C/C++ programs to MLIR. Thus, Bondhugula's hand-crafted input benefits from MLIR passes that are unreachable for a C/C++ end-to-end compilation path. These MLIR passes are also not reachable to compile `Eigen`'s and `BLAS`'s code, our baselines. The experimental results in this Section indicate that the macro-level algorithm and the generic-lowered micro kernel produce code that reaches comparable performance to Bondhugula's approach relative to `BLAS` [10]. Moreover, the MMA lowering reaches up to 96% of `BLAS`'s peak performance, while Bondhugula's reached only 91% of `BLAS`'s performance on a Coffee Lake Intel machine with his MLIR code that employes a target-specific vectorization pass [10].



(a) Intel® x86 (b) AMD® x86 (c) IBM® Power9™

Figure 4.4: Speedup over `PLuTo` for small SGEMM in each platform.



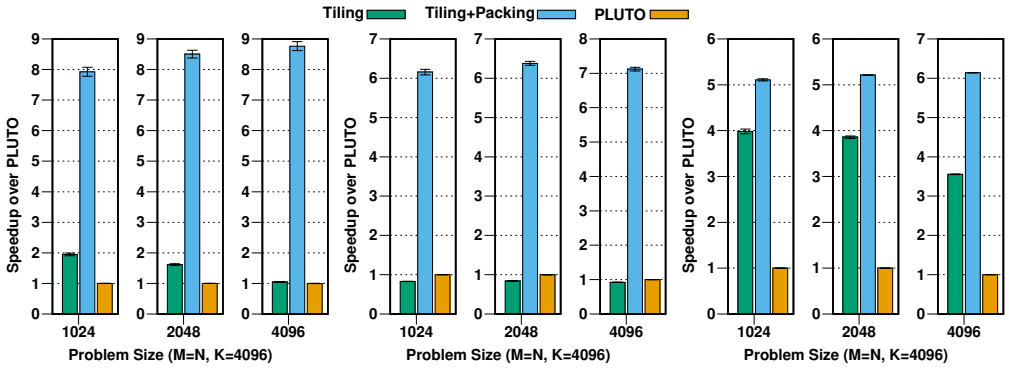
(a) Intel® x86 (b) AMD® x86 (c) IBM® Power9™

Figure 4.5: Speedup over `PLuTo` for medium SGEMM in each platform.

performs best overall as the matrices fit well in the memory hierarchy. Given that small matrices fit in cache, `Tiling+Packing` only adds overhead due extra memory movent for packing input matrices, performing worse than `Tiling`, but still better than `PLuTo`. In addition, `Tiling` is aware of the vector unit capacity in the target architecture and generates a micro-kernel that fully utilizes the vector unit. `PLuTo` performs poorly because its auto-tiling mechanism generates innerloops with conservative tiling sizes which do not saturate the vector unit capacity.

The graphs for medium and large SGEMMs do not include results for `Intrinsic` because the LLVM matrix-multiply intrinsic is designed for small kernels and completely unrolls the loops. Invoking that intrinsic with large dimensions leads to prohibitive compilation times.

Figure 4.5 shows speedup results for medium SGEMM sizes. Overall, `Tiling+Packing` performs best across all platforms. Medium-sized matrices



(a) Intel® x86 (b) AMD® x86 (c) IBM® Power9™

Figure 4.6: Speedup over `PLuTo` for large SGEMM in each platform.

still fit in the low-level caches (L2 and L3), however not in L1. As matrices become larger, they span across multiple memory pages. Therefore, the data reorganization performed by packing improves the utilization of the memory hiearchy by increasing data temporal locality and reducing both TLB and page-faults. `PLuTo` does not employ packing and thus continues to perform poorly for medium-sized matrices. On AMD®, `Tiling` is still compatitive with `Tiling+Packing` due to the larger caches and lower cache access latency. Nevertheless, as Figure 4.6 shows, with large SGEMM sizes that no longer fit the low-level caches (L2 and L3), the performance between `Tiling` and `Tiling+Packing` widens. On Power9™, `Tiling` does not have its performance degraded as much as on Intel® and AMD® for large problems due to large cache line sizes on PowerPC™. However, `Tiling+Packing` remains the best strategy for large SGEMMs due to higher data temporal locality and significantly smaller TLB and page-faults in contrast to `Tiling`.

LLVM has a polyhedral optimizer, Polly [36], that is not enabled by default. If Polly's optimizations are enabled[7] the code runs more than 4.8× slower than `Tiling+Packing` on all three architectures. Polly also has an option to parallelize loops[8]. When this parallelization is run on 20 threads, the resulting code is 1.4× slower than `Tiling+Packing` on Power9™. Unfortunately the SGEMM results computed with Polly-optimized code are incorrect [9]. Thus,

[7] `-mllvm -polly`

[8] `-mllvm -polly-parallel -lgomp`

[9] Polly maintainers are aware of this correctness issue.

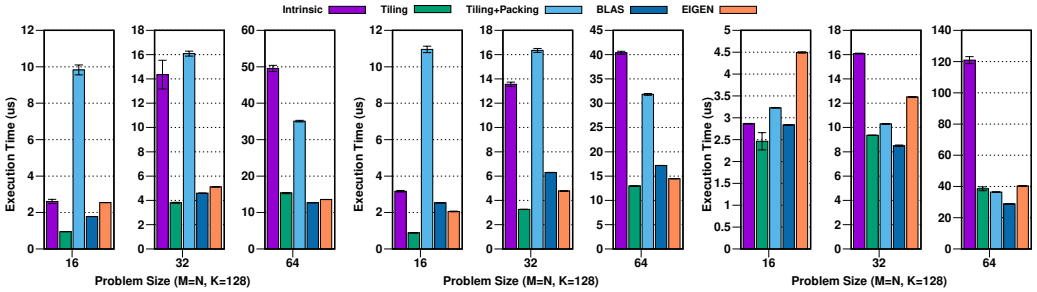
the performance results for Polly cannot be included in the comparison at this time. As reported by Carvalho et al. [16], we also failed to reproduce the performance results reported by Gareev et al.'s recent work [28]. Enabling Polly's GEMM idiom recognition and optimization pass[10] the code runs 4.8× slower than `Tiling+Packing`. Therefore, to the best of our knowledge, this work is the first to present a compiler-only solution that produces performance that is comparable to high-performance libraries (Section 4.2.2) without relying on hand-written assembly micro kernels [10].

LLVM's default lowering of matrix-load intrinsics generates code that spills data from both input operands loaded for the matrix-multiply intrinsic. The maintainers of the matrix intrinsic passes provide an alternative lowering for the matrix-multiply intrisic that bypasses matrix-load intrinsics and generates an unrolled sequence of load-load-multiply instructions with smaller load instructions. This alternative lowering was used as a work-around to eliminate the spill problem on Intel®, AMD®, and Power9™. However, this alternative lowering is currently not modularly integrated into the framework and thus could not be used for the micro-level algorithm. Therefore, the spill problem for `Intrinsic`, `Tiling`, and `Tiling+Packing` on POWER10 was resolved by applying a code transformation after the micro-kernel lowering to sink load instructions – which load the operands of matrix-multiply – closer to their uses – MMA outer-product instructions.

4.2.2 Performance Comparison Against High-Performance Libraries

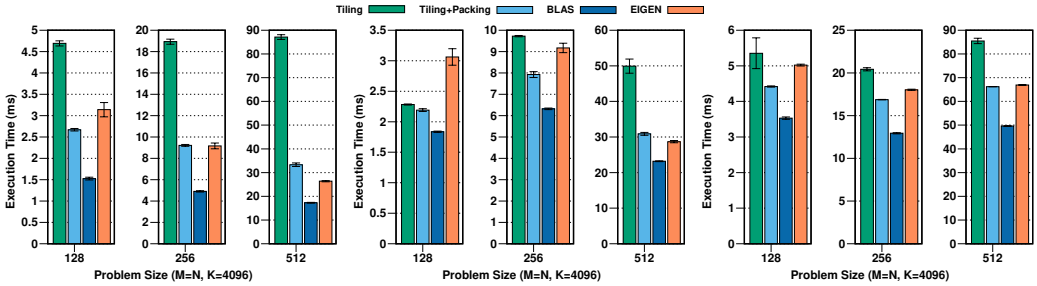
The results in Figure 4.7 indicate that for the smallest SGEMM size the performance of `Intrinsic` is on-par with both `Eigen` and `BLAS`, thus revealing that the backend alone generates efficient code for computing very small SGEMMs. The superior performance of `Tiling` is due to the better utilization of the vector registers in the target architecture. `Tiling` is more than 85% faster than `BLAS` and over 2.6× faster than `Eigen` on Intel® for the problem size $M = N = 16$. On AMD®, `Tiling` is more than 2.3× faster than `Eigen`

[10] `-polly-pattern-matching-based-opts=true`



(a) Intel® x86 (b) AMD® x86 (c) IBM® Power9™

Figure 4.7: Execution time of small SGEMM in each platform.



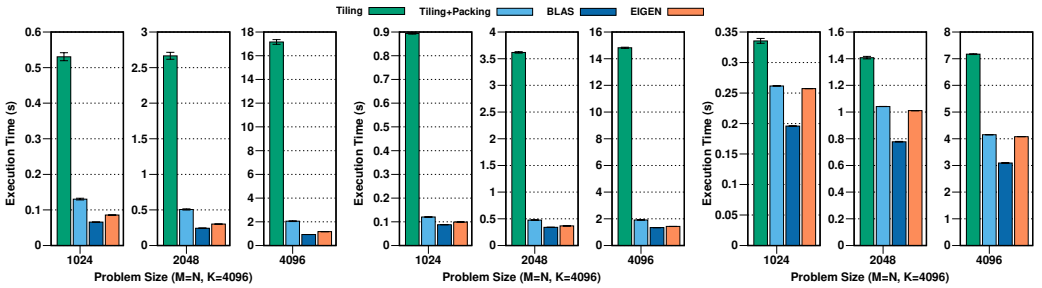
(a) Intel® x86 (b) AMD® x86 (c) IBM® Power9™

Figure 4.8: Execution time of medium SGEMM in each platform.

and 2.8× faster than `BLAS` for the same problem size. `Intrinsic` performs worse in comparison with the libraries as the problem size increases because, lacking tiling, it exhibits lower cache locality. In summary, Figure 4.7 indicates that the compiler-only macro-level approach can surpass high-performance libraries for small problem sizes on multiple platforms.

For medium problem sizes `Tiling+Packing` is the best code-generation strategy overall and it is on-par with `Eigen` across all platforms for the problem size $M = N = 128$. As the size of matrices increases the performance of `Tiling` degrades with respect to both `BLAS` and `Eigen` because LLVM's matrix-load intrinsic lowering results in additional spill code. However, `Tiling+Packing` remains competitive across all platforms by matching `Eigen`'s performance and is less than 80% slower than `BLAS` for $M = N = 512$. As discussed in Section 4.2.1, `Tiling` and `Tiling+Packing` show similar performance due to the larger caches and lower cache access latency on AMD®.

For large matrices `Tiling` performs worst overall because it produces a data layout that leads to more cache misses. `Tiling` increases the cache utilization



(a) Intel® x86 (b) AMD® x86 (c) IBM® Power9™

Figure 4.9: Execution time of large SGEMM in each platform.

within a given dimension but makes cross-dimension accesses expensive because elements of a given column, but different rows, are located on different virtual-memory pages (e.g. 8KB apart on column-major matrix of 2048 columns). The packing data reorganization creates a better element order that increases data locality on both dimensions. Therefore, `Tiling+Packing` performs significantly faster than `Tiling` on Intel® and AMD®. On Power9™, the gap between `Tiling` and `Tiling+Packing` is not as large as in other platforms because Power9™ has larger cache lines (128 bytes) – twice as large as on the other platforms – which, coupled with prefetching, helps to hide memory latency. The `Tiling` performs significantly worse than `Tiling+Packing` on Power9™ as the problem size increases due to increased Translation Lookaside Buffer misses and page-faults. A similar pattern appears on both Intel and AMD machines, where `Tiling` suffers memory penalties as the problem size increases, leaving `Tiling+Packing` as the best performing code-generation strategy overall.

Compiling the largest SGEMM with `Clang -O3`[11], the state of the art when using `Clang` prior to this work, cam result in a code that is more than 68× slower than `BLAS`. This work significantly reduces this performance gap. For the largest SGEMM size ($M = N = 4096$), `Tiling+Packing` is slower than `BLAS`: over 2.2× slower on Intel® and 43% slower on AMD®. In comparison with `Eigen`, `Tiling+Packing` is 1.75× slower on Intel and 34% slower on AMD. On Power9™, `Tiling+Packing` is 34% slower than `BLAS` and on-par with `Eigen`. The performance gap between `BLAS` and `Tiling+Packing` is in great part due to a performance gap between the `BLAS` micro kernel and the generic-lowered

[11] `clang -O3 -mcpu=power9 -mtune=power9 -ffast-math -ffp-contract=fast`

micro kernel available upstream in LLVM. The generic-lowered micro kernel is 2.1× slower on Intel®, 70% slower on AMD®, and 30% slower on Power9™ than the micro kernel from `BLAS`.

Portability is a key design goal for the macro-level algorithm because it leads to the generation of a micro kernel – for any target architecture available in LLVM – with significant performance, as indicated by the results above. Nevertheless, the generic-lowered micro kernel is not a match for the hand-crafted assembly micro kernel available to high-performance libraries. This performance gap can be reduced — as demonstrated for POWER10 MMA (See Section 4.1.2) in the following Section — by the utilization of a platform-specific micro-level lowering algorithm. This strategy of coupling a target-independent macro-level algorithm for tiling and packing with a target-specific micro-level lowering algorithm can be a roadmap for hardware vendors. The results in the following section lead to a prediction that, once micro-level lowering is available for Intel®, AMD®, and Power9™, the gap between the layered compiler-only solution and high-performance libraries can be bridged.

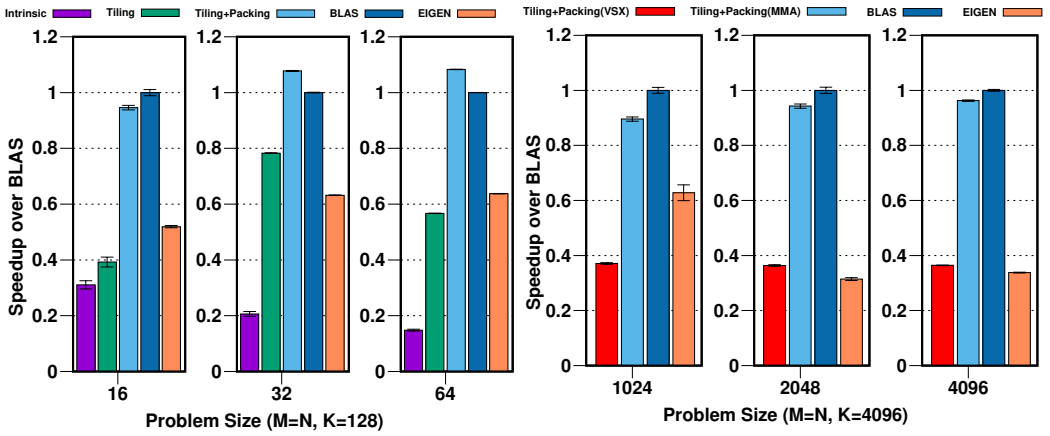
4.2.3 MMA intrinsic

The performance of the compiler-only macro kernel can be boosted by exploiting matrix engines. This study focuses on IBM POWER10's MMA and uses the second-silicon version of POWER10 which has similar hardware characteristics to those that will be featured in commercially available versions but slightly different firmware settings. Therefore, the reported speedups relative to `BLAS` are consistent with the results expected for the commercial version of POWER10. The micro kernels in both `BLAS` and `Eigen` were re-engineered by IBM experts to make efficient use of MMA. The parameters used for these experiments are `mr` = 16, `nr` = 8, and `kr` = 128.

Different than on other platforms, Figure 4.10a shows that for small kernels `Tiling+Packing` results in either better or on-par performance compared to the libraries: `Tiling+Packing` is over 50% faster than `Eigen` for the problem size $M = N = 16$ and over 83% faster for $M = N = 32$ and $M = N = 64$. `Tiling+Packing` achieves up to 10% more performance than `BLAS` for

SGEMMs of size $M = N = 32$ and $M = N = 64$. With MMA, as GEMM is implement with outer products instead of inner products, columns of matrix A are multiplied by rows of matrix B producing an accumulator result that has partially computed column elements of matrix C – one per VSR associated with the accumulator. Therefore, the best access layout for the MMA builtin is both A and C in column-major and B in row-major order. However, all input matrices are stored, and thus accessed, in column-major order. Therefore, loaded tiles of matrix B need to be transposed prior to calling the MMA builtin. For `Intrinsic` and `Tiling`, this means additional vector shuffle and merge instructions in the micro-kernel code. The extra instructions in the case of `Tiling+Packing` are not generated as part of the micro level but as part of the macro level algorithm that packs the matrices. With fewer instructions, the micro-kernel code generated with `Tiling+Packing` runs faster and exhibits better instruction-cache utilization. In addition, the packing loads act as prefetching loads and reduce access latency for the operands of outer-product instructions. Moreover, calling a library incurs in performance overheads that are more noticeable on smaller problem sizes. These results indicate that the proposed macro-level approach coupled with an MMA-specific lowering of `llvm .matrix.multiply` can provide better performance than two state-of-the-art libraries.

Figure 4.10b contrasts the performance of the generic-lowered code, which uses VSX instructions on POWER10, with the performance of the new MMA-specific lowered code presented in this chapter (Section 4.1.2). The MMA solution is over 2.6× faster than the VSX solution for the largest SGEMM. The conclusion is that a target-specific lowering can outperform `Eigen`, when both use MMA instructions, for the SGEMM sizes $M = N = 2048$ and $M = N = 4096$. In fact, `Tiling+Packing` with VSX matches the performance of `Eigen` with MMA, indicating that the macro-level algorithm made better tiling and packing decisions with compile-time information than `Eigen`. `Tiling+Packing` provides over 2.6% better performance than `Eigen` for the largest SGEMM. Furthermore, the MMA-specific lowering algorithm generated code that closely matches `BLAS` performance (up to 96%), which itself achieves



(a) Speedup over `BLAS` with MMA. (b) VSX vs. MMA performance.

Figure 4.10: (a) Speedup over `BLAS` of small SGEMM on POWER10 with MMA; (b) Contrasting VSX and MMA performance of SGEMM kernel on POWER10.

almost 50 flops/cycle (almost 80% of peak). In essence, a compiler-only layered approach built around a compiler intrinsic for matrix multiplication can achieve comparable or better performance than state-of-the-art libraries. In addition, having a target-specific lowering for the matrix-multiplication instrinsic is key to make best use of vector/matrix units, the higher cache locality, and better utilization of the memory hierarchy delivered by the macro-level algorithm.

4.3 Additional Opportunities

This groundwork implementation of the layered approach in LLVM creates opportunities that will benefit other BLAS kernels and other specialized architectures.[12]

4.3.1 Macro-level strategy for other BLAS kernels

Algorithm 3 describes general techniques for blocking, tiling, packing and intrinsic invocation that can be used for other BLAS matrix operations. As an example, consider SYR2K, which computes either the lower or upper triangular half of $C \leftarrow \alpha \times A \times B^T + \alpha \times B \times A^T + \beta \times C$. Matrix C is symmetric while matrices A and B are general. High performance implementations of SYR2K

[12]The implementation will be openly available as an artifact.

partition the matrix C into blocks and use a pair of GEMM operations to update each block. Both the normal (non-transposed) and transposed versions of matrices A and B are needed.

An equivalent to Algorithm 3 for SYR2K, will use two calls for packing matrix B (line 3) and two calls for packing matrix A (line 5). In each case, one of the calls produces a packed version of a block of the matrix and the other call produces a packed version of the transpose of a block with different blocks used for the normal and transposed versions. In the inner loops, the algorithm loads two tiles of B (line 10) and two tiles of A (line 11), one tile from the normal block and the other from the transposed, each with different tiles used from the normal and transposed versions. Finally, the actual computation (line 12) requires two calls to the `llvm.matrix.multiply` intrinsic. This approach reuses the tiling and packing strategies and allows the use of blocks and tiles of different sizes for matrices A and B to achieve better cache utilization.

4.3.2 Targetting other matrix engines

POWER10 MMA is but one of the emerging enhancements conceived to accelerate matrix operations in modern CPUs and GPUs. For CPUs two other architectural extensions have been announced: Intel's AMX [53] and Arm's ME [5].

Each extension has its own unique characteristics, including the list of supported data types, the set of registers used to operate on matrices, and basic computational operation. Table 4.2 shows a brief comparison of the three extensions. It seems that AMX is more targeted to image-processing and deep learning applications, while both MMA and Arm's ME can also handle scientific applications.

MMA and Arm's ME are tightly integrated in the processor core while AMX instructions are executed in a more loosely coupled accelerator (TMUL). An Intel host CPU communicates with an AMX unit via a queue of tiles and accelerator commands [53]. Tile commands load/store tile register data from/to memory addressed by host CPU registers. All pointer arithmetic and loop control instructions run in the host CPU. Coherence between accesses from the

Table 4.2: Matrix multiplication extensions comparison.

Extension	Types	Registers	Operation
IBM® MMA	`i4`, `i8`, `i16` `bf16`, `f16`, `f32`, `f64`	VSRs + ACCs	rank-k update $k = 1, \ldots, 8$
Intel® AMX	`i8`, `bf16`	Separate register file 16×64-byte tiles	tile multiplication
Arm ME	`i8`, `bf16`, `f32`, `f64`	Vector registers only	2×2 matrix operations

AMX unit and access from the host CPU is maintained at the main memory level instead of at the cache level [53].

The approach described in this chapter can be used with these other matrix engines. The macro-level algorithm described in Section 4.1.1 is general and can be lowered to multiple targets supported by LLVM's backend. The compiler designer must tailor a few lowering operations when moving to a new target. In particular, the load/store operations for packing (Algorithm 3, lines 3 and 5) and for loading and storing tiles (Algorithm 3, lines 10, 11, 14, and 19) must be lowered to architecture-dependent instructions by the LLVM backend. Lowering to Arm's ME and IBM®'s MMA is straightforward because both matrix engines are tightly coupled accelerators and rely on host CPU vector registers to store input and output data. On AMX however, the lowering of load and store operations use the`TILELOAD*` and `TILESTORE*` AMX instructions. These cause data movements between AMX's tile register and host memory. There are no instructions to move data between AMX's tile registers and host vector-registers (e.g. AVX-512 registers). Therefore, for both GEMM and SYR2K, where partial products are scaled by α and β (Algorithm 3, lines 17 and 16), tile-register data would have to be stored back to memory and loaded into host vector register for the scaling computations.

4.4 Concluding Remarks

This work presents a robust solution to the problem of generating efficient code entirely within a compiler targeting multiple architectures, consisting of implementing, in the widely used LLVM compilation framework, the layered approach broadly used in specialized libraries. A key insight was to create a

parameterized algorithm for the tiling and packing layer that only requires the compiler to read the effective sizes of the caches from an existing LLVM pass to determine the appropriate sizes for blocks. In this approach, a target-specific compilation can use the size of the register file for the target architecture to decide on the most appropriate size for tiles. Similar to the layered approach used in libraries, the goal of packing is to lay out the tiles in memory in the order in which they will be accessed during the computation to increase locality. Another essential insight was to use the standard LLVM matrix multiplication intrinsic as the interface between the macro kernel and the micro kernel. This way, for any target architecture, the specialized code generation at the micro-kernel level only needs to be done once and its performance advantages will benefit any code generation path that uses the same intrinsics. The performance evaluation, including machines with and without matrix engines, demonstrates the modularity of the design and reveals significant performance gains from the layered approach in multiple architectures. The experimental evaluation indicates that the macro-level algorithm, coupled with a generic intrinsic lowering, achieves more than 22× better performance than PluTo, a widely used compiler-only polyhedral optimizer. The new compiler-only approach generates code that matches `Eigen` performance and is only 34% slower than `BLAS` on Power9™. An MMA-specific implementation results in more than 2.6× the performance of the VSX micro kernel, is over 83% faster than `Eigen`, and achieves up to 96% of `BLAS` peak performance for large SGEMMs, even when these libraries are engineered to also benefit from MMA.

Chapter 5

YaConv: Convolution with Low Cache Footprint

Convolutional neural networks (CNNs) deliver reliable solutions for the problems of image classification, speech recognition, recommendation, and language translation. Software frameworks such as Caffe, Tensorflow, and PyTorch have emerged to support the increasing variety of CNNs on multiple hardware architectures. Most of these frameworks introduce a middle-layer representation for the network primitives that are efficiently implemented in high-performance numerical libraries [52], [70], [94].

Training and running a CNN is a computationally-intensive task with convolution layers accounting for over 80% of the total CPU inference time on networks VGG-16, Resnet-52 and GoogLeNet. More than three-quarters of the CNN inference market relies on CPUs because of their low inference latency [85]. General matrix multiplication (GEMM) is the most-used primitive in high-performance libraries. Convolution is typically computed by performing the `im2col` transformation on the input image and calling a library GEMM routine [19]. Other approaches suggest reducing the memory footprint of `im2col` or implementing convolution through efficient architecture-specific assembly kernels. [4], [20], [30], [31], [58].

The convolution algorithm presented in this chapter targets memory hierarchy optimization on CPUs. **The core idea of** `YaConv` is to pack the input image into an L3-cache-resident buffer, preload a smaller chunk of the buffer into L1 cache, and use this image chunk for computation with all L2-

cache-resident filter elements before switching to other image elements. To our knowledge, `YaConv` is the first convolution algorithm that implements all of the following:

1. uses unchanged GEMM microkernel from a high-performance library;
2. integrates domain-specific packing of elements the into cache with the calls to GEMM microkernels;
3. avoids unnecessary additional copies of input-image elements.

The two last points are important distinct design decisions. Several methods have been proposed to eliminate or reduce the copy of the image tensor in memory [4], [20], [58]. However, unlike `YaConv`, they do not address the cache reload issue encountered while calling a library GEMM routine multiple times on the same elements.

`YaConv` successfully repurposes GEMM building blocks to perform convolution. It aims to compute convolution at a performance level that is close to the machine's peak without requiring the writing of new assembly kernels. There are several ways to implement a convolution algorithm with the same constraints. The version implemented and evaluated in this chapter is the most promising in terms of performance on the actual layers found in PyTorch.

For some layers, `YaConv`'s performance is on par with a library GEMM routine, which is around $80 - 90\%$ of the machine's theoretical peak performance [35], [70], [94]. In comparison with another architecture-independent solution, `im2col` convolution, `YaConv` is 24% faster, measured as the geometric mean of the speedup (w.r.t. `im2col`) over 73 layers taken from real CNN models. Moreover, `YaConv` achieves this level of performance using $10\times$ less memory than `im2col` convolution, requiring only a small buffer space.

An experimental study based on varying input image sizes confirms that the performance of `YaConv` is dependent on architecture-specific parameters within the GEMM microkernel. The results of this study point to ways to reduce this sensitivity by tuning the image height in the intermediate layers of certain CNNs. Cache utilization evaluation on the range of inputs reveals that

two parameters — the number of output channels (M) and image height (H) — affect the performance of `YaConv` the most, with consistent speedup over the baseline when $M < 500$ or $H > 20$.

5.1 Cache Inefficiencies of Previous Algorithms

Figure 2.2b shows how a naive algorithm computes convolution $I_{5\times5\times2} \otimes W_{3\times3\times2\times M} = O_{3\times3\times M}$. For this example, the naive method iterates over image patches of dimensions $[3 \times 3 \times 2]$ and computes the sum of products between each input image patch and each filter. The highlighted output elements in Figure 2.2b are computed using all weight elements in the filter and the input elements surrounded by dashed lines. The depth dimension of the selected output elements (output channels) corresponds to the number of filters in the weight tensor.

The naive approach underutilizes the available vector units on a CPU and suffers from poor cache locality of the patch elements. In Figure 2.2b, every $F_w \cdot C = 3 \cdot 2 = 6$ elements of each image patch lie consecutively in memory. However, two consecutive rows within the same patch lie at an offset of $(W - F_w) \cdot C = (5 - 3) \cdot 2 = 4$ elements in memory, as each patch of size $I_{3\times3\times2}$ is a part of the entire image $I_{5\times5\times2}$. This causes loading of cache lines and populating TLB entries for the elements not in the order they are accessed during computation.

5.1.1 Convolution With `im2col` Transformation

`im2col` addresses the problems of the naive approach by placing the patch elements adjacently in memory and calling a performant GEMM routine to compute the output. Figure 5.1 provides an example convolution $W_{3\times3\times C\times M} \otimes I_{5\times5\times C} = O_{3\times3\times M}$ computed through the `im2col` + GEMM path. The whole algorithm is broken into four steps, indicated with red circles. Along with the description of the `im2col` transform, Figure 5.1 shows data movement in the cache-resident buffers of the GEMM routine.

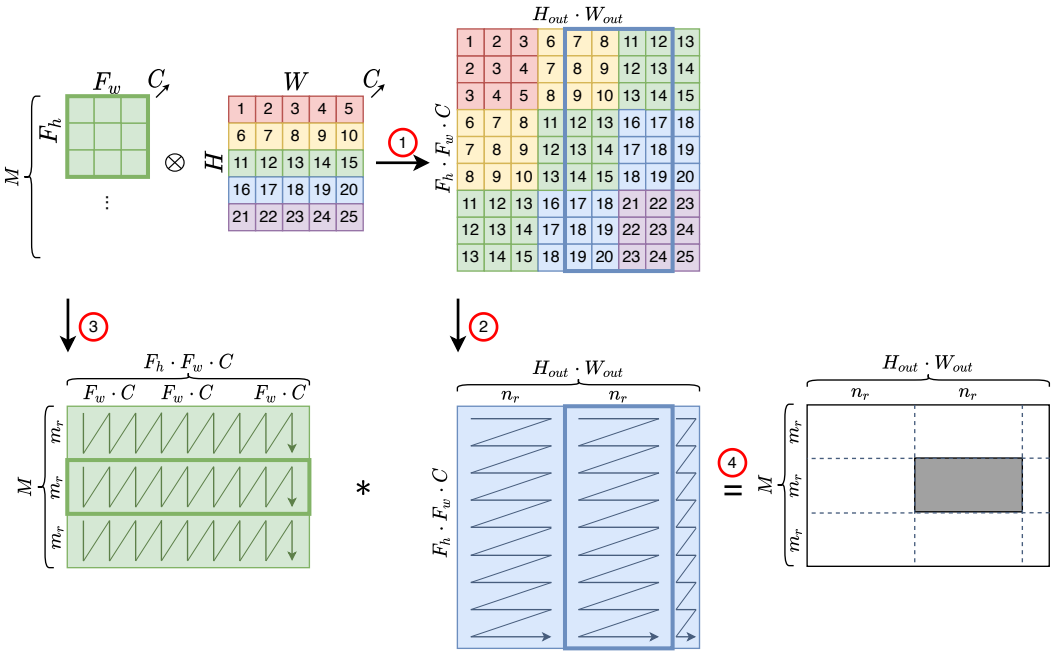


Figure 5.1: `im2col` convolution

Step (1) in Figure 5.1 demonstrates how `im2col` copies the input tensor into a patch buffer as an $[F_h \cdot F_w \cdot C] \times [H_{out} \cdot W_{out}]$ matrix. Each column in this matrix is a flattened patch of the input of size $F_h \times F_w \times C$. There are $H_{out} \cdot W_{out}$ columns in the `im2col` buffer corresponding to the output image of size $H_{out} \times W_{out}$. Computing a GEMM between the reshaped weight tensor as a matrix $[M] \times [F_h \cdot F_w \cdot C]$ and the `im2col` matrix produces a matrix of shape $[M] \times [H_{out} \cdot W_{out}]$. The output of GEMM is the output tensor $H_{out} \times W_{out} \times M$ stored as a column-major matrix of leading dimension M in memory.

Step (2) in Figure 5.1 shows the movement of the elements of the `im2col` buffer during the GEMM call. In the GEMM implementation, the second matrix is packed to an L3-cache-sized buffer in a layout that facilitates L1 cache and register reuse by the outer product microkernel (Section 2.1). The blue arrows within the packed `im2col` buffer demonstrate the order of the elements in memory, with each arrow's length capped at n_r — an architecture-dependent factor introduced in Section 2.1.

Step (3) in Figure 5.1 shows how the GEMM routine packs the weight tensor into an L2-cache-resident buffer as an $[M] \times [F_h \cdot F_w \cdot C]$ matrix. Packing

the weight tensor after the `im2col` buffer ensures that the packed weight buffer elements are in L2 cache and the majority of the packed `im2col` buffer elements are in L3 cache (except for those that were evicted during the weight copy). Similarly to the packed `im2col` buffer, the green arrows of length m_r (Section 2.1) show the memory layout of the elements in the weight buffer.

In step (4), the outer product GEMM microkernel multiplies the tiles $[m_r] \times [F_h \cdot F_w \cdot C]$ of the packed weight and $[F_h \cdot F_w \cdot C] \times [n_r]$ of the `im2col` buffer. The result of each microkernel call is stored in the output as a block of size $m_r \times n_r$ at the corresponding tile offset in the packed buffers.

The GEMM routine applies tiling along each matrix dimension to ensure that each packed buffer fits in the respective cache level, in the case when the weight tensor and/or the `im2col` matrix are larger than L2 and/or L3 cache. Placement of the packed buffers into L2 and L3 cache is ensured by the order of the packing steps. By packing the `im2col` buffer before the weight tensor, the GEMM routine ensures that the weight elements are not evicted from L2.

Weight and `im2col` buffer elements are streamed from the respective packed weight and packed `im2col` buffers that reside in L2 and L3 cache. For instance, highlighted tiles in Figure 5.1 — one from the packed weight buffer and one from the patch buffer — are multiplied to produce the block of output shown as a grey rectangle. Tile offsets within the weight and patch buffers directly translate into vertical and horizontal (on Figure 5.1) block offsets in the output. Parameters m_r and n_r are optimized for L1 cache and register reuse within the microkernel.

5.2 YaConv

Two principles are at the core of the new convolution algorithm: the algorithm should not require redundant copies or loads of input elements in cache and the algorithm must use unaltered GEMM microkernels. We follow the CPU cache utilization guidelines presented in [35]. `YaConv` introduces a new iteration pattern for convolution and controls packing of the input tensor

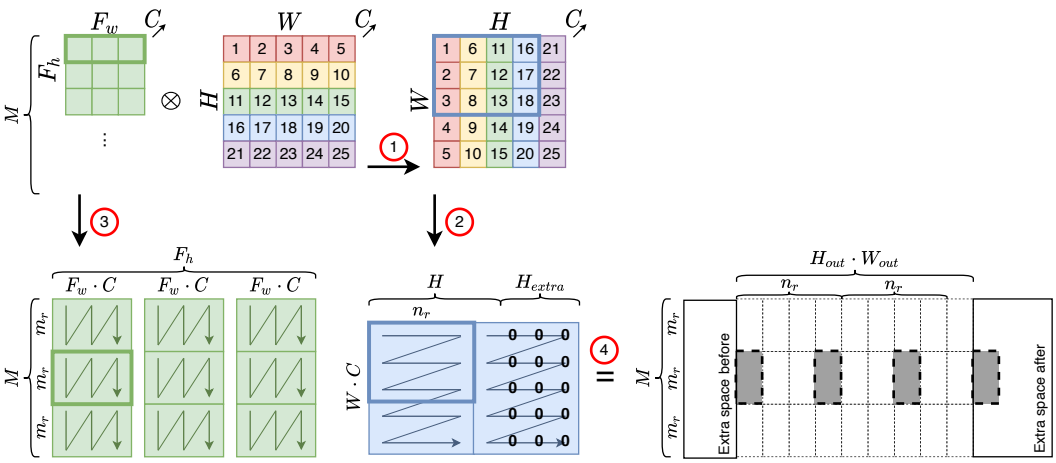


Figure 5.2: The novel `YaConv` algorithm

elements into the cache.

In naive convolution, two loops iterate over the spatial dimensions of the input image and compute the sum of products between the weight tensor and each image patch (Figure 2.2b). In `YaConv`, these sums do not happen in the same loop iteration. Instead, the `YaConv` computes one-row convolutions between a selected row of each filter and the corresponding image patches.

Figure 5.2 demonstrates how `YaConv` computes the same convolution as in Figure 5.1 using packing and outer-product microkernels from a library GEMM. In step (1), `YaConv` reshapes the input tensor as a column-major matrix $[W \cdot C] \times [H]$. This step does not incur any data copy overhead because the memory layout of such a matrix matches the layout of the input tensor $I_{H \times W \times C}$.

Step (2) in Figure 5.2 shows how `YaConv` packs the input tensor into an L3-cache-resident buffer. Zeroed-out elements in the right tile of size $[W \cdot C] \times n_r$ artificially extend the size of the packed input tensor buffer. The reason for doing so lies within the GEMM microkernel — it performs poorly when the matrix sizes are not multiples of n_r. By first packing the input tensor, we bring the input tensor elements to all cache levels, ensuring that the buffer is in L3 cache.

In step (3), `YaConv` packs the weight tensor as F_h separate matrices of size $M \times [F_w \cdot C]$ into an L2-cache-resident buffer. This copy operation might

evict some input tensor elements — as it happens in the traditional GEMM algorithm — but ensures that $M \cdot F_w \cdot C$ weight tensor elements are loaded to L2 cache. To perform partial convolutions for each filter row, one $M \times [F_w \cdot C]$ matrix is packed in the weight buffer at a time.

In step (4), `YaConv` computes each convolution $W_{1\times F_w \times C \times M} \otimes I_{H \times W \times C} = O_{H_{out} \times W_{out} \times M}$ as W_{out} GEMMs with sizes $[H] \times [F_w \cdot C] \times [M]$. Having packed the input tensor as contiguous tiles of size $[W \cdot C] \times [n_r]$, `YaConv` passes portions of size $[F_w \cdot C] \times [n_r]$ of the packed input buffer as a second operand to the GEMM microkernel — one of such portions is highlighted by the blue rectagle in Figure 5.2. The first operand of the microkernel call is a tile of size $[m_r] \times [F_w \cdot C]$ from the packed weight buffer.

Each GEMM with sizes $[m_r] \times [F_w \cdot C] \times [n_r]$ computes n_r output elements for m_r output channels. All of these elements correspond to the same column of the output image (W_{out}-dimension), given by the vertical offset of the tile of size $[F_w \cdot C] \times [n_r]$ within the packed input tensor. One GEMM call computes the result of applying one row of m_r filters to n_r rows of the input image at the same column offset. The corresponding weight and input tensor elements are highlighted by green and blue rectangles in Figure 5.2. Each result of size $m_r \times n_r$ is placed in the output array (column-major $[M] \times [H_{out} \cdot W_{out}]$ matrix) as n_r columns at stride $W_{out} \cdot M$, because every GEMM call corresponds to applying one filter row over n_r input image rows at a fixed column.

5.2.1 Extra Memory Usage

Because `YaConv` computes GEMM between every possible combination of the weight and image tiles, and some of these elements are not part of the convolution result, they are not accumulated in the output. In the example in Figure 5.2, the first filter row is multiplied with elements of the blue portion of the input image as part of the GEMM microkernel call with width $n_r = 4$. However, only the first three columns of this highlighted image area should be multiplied with the first filter row and the result of this multiplication to be stored in the output tensor. The result of the dot product between the elements $16, 17, 18$ of the input tensor and the first filter row is a part of applying the

filter on the image when the last filter row is outside of the image bounds.

One way to discard these elements is to adjust the width of every GEMM microkernel call to smaller values than n_r, leading to significant performance degradation. Instead, `YaConv` reserves a larger buffer for the output elements than the $H_{out} \cdot W_{out} \cdot M$ elements needed by convolution. The actual output tensor $H_{out} \times W_{out} \times M$ is located at an offset within this buffer space, while the extra space is used for the spillover elements of the GEMM calls.

The output buffer contains the output array and two extra parts *before* and *after* the actual output tensor. For a given problem, `YaConv` uses space for $(F_h - 1 - P_h) \cdot W_{out} \cdot M$ extra elements before the output array. Because the image buffer is enlarged to the optimal microkernel size and to account for the spillover results from GEMM, `YaConv` also reserves some space for the elements immediately after the actual output. In Figure 5.2, H_{extra} is the H enlargement required until the next multiple of n_r. Therefore, extra space required after the output array is $(H_{extra} + F_h - 1 - P_h) \cdot W_{out} \cdot M$, where the second part comes from the GEMM spillover elements. In total, `YaConv` requires $(H_{extra} + 2 \cdot (F_h - 1 - P_h) \cdot W_{out} \cdot M$ extra space, at that $H_{extra} < n_r$. Asymptotically, extra space complexity for `YaConv` is $O((F_h - P_h) \cdot W_{out} \cdot M)$, which is sublinear on the output size $H_{out} \cdot W_{out} \cdot M$.

5.2.2 Tiling and Block Sizes

In real applications, weight and image tensors are large enough to not fit in the cache. We add loop tiling to our algorithm to improve cache locality and TLB entry usage, as suggested by Goto and Geijn. The conventional GEMM algorithm [35] uses three cache block sizes MC, KC, NC for two packed buffers $\tilde{A}_{MC \times KC}$ and $\tilde{B}_{KC \times NC}$:

1. KC is calculated to fill L1 cache with tiles $m_r \times KC$ and $KC \times n_r$ of the operands of each microkernel call

2. MC is set to fill the L2 cache with $MC \times KC$ elements of the packed buffer $\tilde{A}$

Architecture	Clock, GHz	L1, KiB	L2, KiB	L3, MiB	GEMM tile size
Intel® Cascade Lake™	3.5	32	1024	36 / 24	32×12
AMD® Zen 2™	2.0	32	512	16 / 4	6×16
IBM® POWER10	4.0	32	2048	64 / 8	8×16
Intel® Haswell	2.7	32	256	30 / 12	6×16

Table 5.1: Clockrate, cache sizes and output tile dimensions of the GEMM microkernel of the machines used for the experiments. L1 and L2 cache sizes are per core. L3 size is followed by the number of cores sharing L3 cache.

3. NC determines the size of the buffer $\tilde{B}$ that resides in L3 cache

High-performance BLAS libraries set these sizes more conservatively as temporary variables and output tile $m_r \times n_r$ take some L1 cache space, and TLB is typically a more limiting factor than L2 cache [35], [70], [94].

The sizes for weight and image buffers in `YaConv` are taken from the BLIS implementation of the GEMM routine. The order of calls to packing routines determines the cache-level residence for each buffer. `YaConv` packs a portion of the image in the outermost loop, thus loading its elements to all cache levels. The packing of weight elements into the buffer follows image packing. Such an order, coupled with adequate tile sizes, ensures that the weight elements will not be evicted from L2 cache by the image elements during packing. Two innermost loops around the microkernel (with steps n_r and m_r) iterate over L1-sized tiles of weight and image buffers and call the microkernel code to compute partial results.

Additionally, when $W = F_h = F_w = 1, P_h = P_w = 0$, the weight and image tensors can be thought of as matrices and convolution degenerates into a GEMM with sizes $M \times C \times H_{out}$. The loops over F_h and W_{out} contain only one iteration and `YaConv` becomes the conventional matrix multiplication algorithm.

5.3 Comparing `YaConv` with `im2col` on Multiple Machines

The experimental results in this section indicate that:

1. `YaConv` outperforms the `im2col` baseline on PyTorch layers by 23-25% on multiple architectures.

2. The superior performance of `YaConv` is explained by better L3 cache usage. Moreover, in most cases, `YaConv` reduces L1 cache usage as compared to `im2col`-based convolution.

3. As expected, the performance of `YaConv` compares unfavourably with `im2col` for small image heights H which are not a multiple of architecture-dependent GEMM microkernel sizes. Better performance for `YaConv` can be achieved by adjusting the image size to match architecture-dependent values.

Parameter(s)	Common values
H, W	$14, 7, 28, 56$
F_h, F_w	$3, 5$
C	$64, 192, 32, 128$
M	$128, 256, 64, 192$

Table 5.2: Values for selected convolution parameters from 218 layers in PyTorch, from most to least common.

5.3.1 Experimental Methodology

Table 5.1 provides hardware information about the four machines used for the experiments. The cache sizes are given per node, *i.e.* L3 cache is shared among some cores on each platform. Each binary runs the same convolution on a batch of N images, where N is adjusted to ensure that each execution lasts at least 1 second on a 100 GFLOPS machine. FLOPS are calculated as the number of single-floating-point operations, given by $2 \cdot N \cdot H \cdot W \cdot C \cdot F_h \cdot F_w \cdot M$, divided by the wall clock time of the respective convolution routine on the whole image batch. The results presented for each experiment are mean values of ten runs. Unless explicitly specified, the relative standard deviation observed is less than 5%.

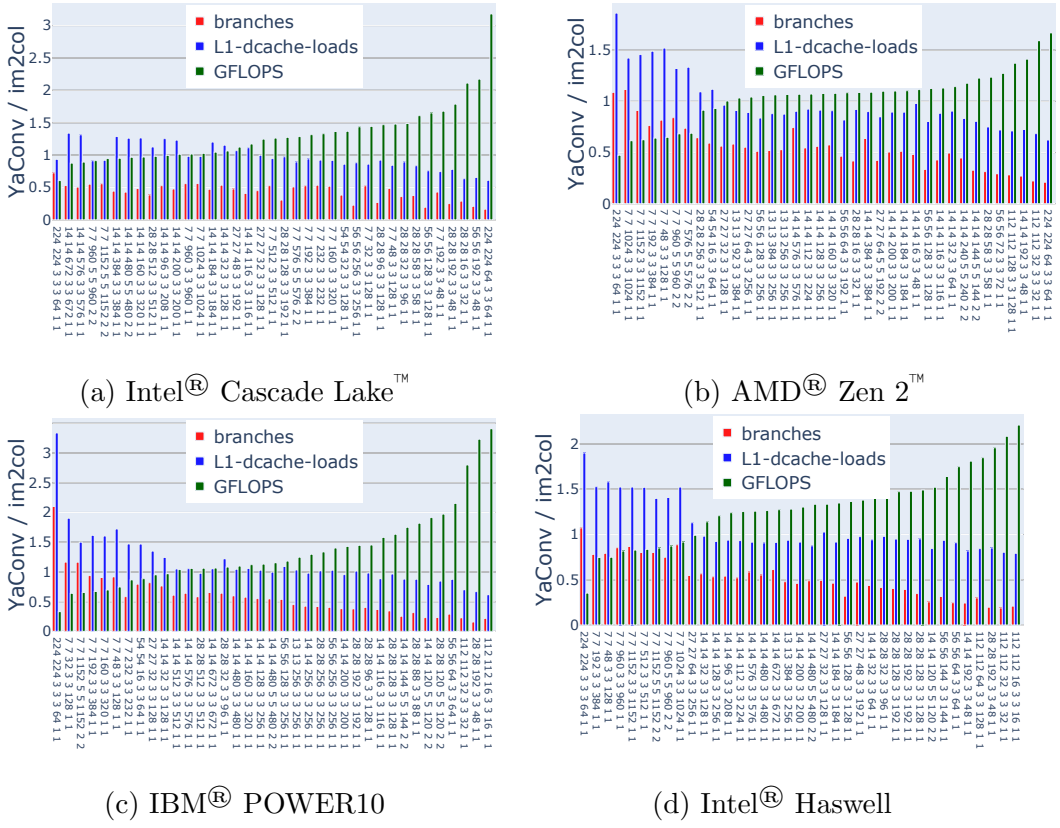
Benchmarking cache performance is difficult because of the complex hierarchy of modern CPU memory systems. Although `im2col` does extra work copying input image elements to another buffer, it loads the elements into the cache and populates the TLB entries. This data preparation by `im2col` reduces data access time within the packing routines of the library GEMM. Thus, for a fair comparison, this experimental evaluation compares the performance of the whole `im2col`-based convolution routine with `YaConv`. There should be fewer accesses to the last levels of the memory hierarchy by `YaConv` because `YaConv` loads each input image element exactly once into an L3-cache-resident buffer. Both `YaConv` and `im2col`-based convolution implementations allocate temporary space once for the whole batch of images.

The implementation of `im2col` from Caffe is used for the baseline, followed by a call to BLIS GEMM [56]. BLIS[1] was built using the default library-provided flags for each platform and the same flags were used to build the microkernels for integration into `YaConv`. All platforms run a 64-bit Linux kernel and all benchmarks are compiled using gcc with -O2 with -mtune=native. **perf** was used to collect cache and TLB counters [91].

5.3.2 Performance on PyTorch Layers

This evaluation uses convolutional layer parameters from thirteen pre-trained CNN models in Torchvision, which is a part of the PyTorch project [73]. Out of a total of 661 layers, 400 layers are a special case of convolution with filter size 1, which both PyTorch and Tensorflow compute as a direct call to the library GEMM. Also, `YaConv` cannot handle convolutions with non-unit strides because of the element ordering restrictions imposed by the GEMM microkernel. Thus, another 42 non-unit-stride layers were eliminated leaving 218 layers with unit stride and filter size greater than 1. Some of these layers have the same geometry: 73 unique layers can be used for the evaluation of `YaConv`. The most common values for H, W, F_h, F_w, C, M are provided in Table 5.2.

[1]BLIS version b3e674db3c05ca586b159a71deb1b61d701ae5c9



(a) Intel® Cascade Lake™

(b) AMD® Zen 2™

(c) IBM® POWER10

(d) Intel® Haswell

Figure 5.3: Ratios for L1 cache, branches and GFLOPS between `YaConv` and `im2col` on layers from PyTorch across four machines

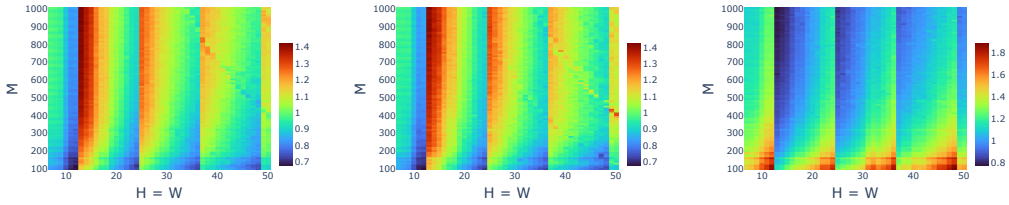
Figure 5.3 presents the ratios between `YaConv` and `im2col` for the following measurements provided by `perf`:

1. number of L1 cache accesses;
2. number of total branch instructions executed;
3. total routine GFLOPS.

Layers are shown on the x-axis in the format $H\ W\ C\ F_h\ F_w\ M\ P_h\ P_w$ and are sorted by the GFLOPS ratio between `YaConv` and the baseline. To improve visibility, the graph presents only half of the 73 layers. Layers are selected by choosing every second layer from the sorted list. Figure 5.3 indicates that `YaConv` reduces the number of branches taken by the `im2col` convolution algorithm. For all machines, the ratio of L1 cache loads between the two algorithms decreases with a larger speedup of `YaConv` over `im2col` convolution, pointing to increased reuse of elements in L1 cache as the source of the observed speedup.

The reduction in the number of branches taken in comparison with `im2col`-based convolution is due to the elimination of the `im2col` transform. `YaConv` does not make a copy of an image element for each of the filter elements. `YaConv` loads each image element into L3 cache exactly once, resulting in fewer L3 cache loads and misses. However, for some layers, the L3 cache performance measured in the experiments does not support this hypothesis. Cache design in some architectures is complex and L3 performance requires further investigation. Additionally, the slowest layers in Figure 5.3 are also the ones where `YaConv` accesses L1 cache more than the baseline. Two following sections present a detailed study on Intel® Cascade Lake™, where parameters are varied to identify performance trends. The same studies were conducted on all four machines, exhibiting similar trends in algorithm runtimes and cache utilization. Detailed results are presented for one machine for brevity.

5.3.3 `YaConv` Performance Varies with Image Sizes



(a) L1 cache loads (b) TLB loads (c) GFLOPS

Figure 5.4: Varying image size $H = W$ and number of input channels with fixed $M = 200, F_h = F_w = 3, P_h = P_w = 1$ on Intel® Cascade Lake™

The values of H, W and C in the layers in Figure 5.3 were selected from the real CNNs from PyTorch. A more comprehensive understanding of the performance of `YaConv` can be gained by varying these values. Figure 5.4 presents cache and runtime profile collected on the Intel® Cascade Lake™ machine using the same methodology as in the Section 5.3.2. All subfigures have fixed parameters $C = 300, F_h = F_w = 3, P_h = P_w = 1$ and vary square image sizes $H = W$ and M in the range of values found in actual CNNs. The color of points in the heatmaps represents the value of the ratio $\frac{\texttt{YaConv}}{\texttt{im2col}}$ for each metric.

Two insights can be gained from Figure 5.4. First, the repetitive pattern with step size 12 on the plots confirms that computing full GEMM microkernel on partially-filled tiles significantly affects performance. For this machine, the microkernel uses $n_r = 12$ to maximize the use of the vector unit. Therefore, when H is a multiple of 12, Figure 5.4 shows the fewest L1 cache and TLB accesses and the best runtime for `YaConv`.

Second, `YaConv` performs better with smaller values for M. With the fixed value $C = 300$, relative speedup of `YaConv` over `im2col` convolution gradually decreases until $M = 480$. After that point, the relative performance stabilizes with further increase in M. As shown in Figure 5.6a in the following Section 5.3.4, this effect can be explained by the gradual increase in the number of L3 cache loads that depends on the memory stride M of the weight tensor elements. On the other hand, performance metrics do not show any trend while varying the number of input channels C.

In Figure 5.3a, the worst-performing layers on Intel® Cascade Lake™ have

$H = W = 7, 13, 14$, most of which come from one CNN (GoogLeNet). In `im2col`-based convolution, the width of the image buffer is $H \cdot W$ which includes several full GEMM tiles even for the smallest images $H = W = 7$. In `YaConv`, the same input image is packed into a buffer of width H and padded to the full tile with zeroes. For larger image sizes this padding does not play a big role because the partially-filled tiles are a small portion of the whole computation. However, for small H, the extra zero elements require more usage of the cache and the TLB entries and the result of GEMM for these elements is not used for accumulation of the output. This is confirmed by sharp vertical edges in Figure 5.4a and Figure 5.4b. The performance patterns shown in Figure 5.4 are also observed in other architectures and can be used to improve the performance of `YaConv` by adjusting the image size for layers in the middle of the network according to architecture-specific values.

5.3.4 `YaConv` Improves L3 Cache Performance

The vertical axis in Figure 5.5 presents the ratios of L3 accesses and misses and GFLOPS between `YaConv` and `im2col` collected on Intel® Cascade Lake™. The horizontal axis contains the same layers as Figure 5.3a. While most layers experience a reduction in L3 cache usage, `YaConv` brings an increase in cache accesses and misses for certain sizes of the parameters C and M, *e.g.* 128, 512, 1024. These increases in L3 accesses and misses seem to occur more often when $H = W = 7$. Thus, a parameterized performance study could reveal more information about the correlation of these misses with performance.

The set of experiments shown in Figure 5.6 varies the number of input channels C on the horizontal axis, the number of output channels M on the vertical axes with fixed image and filter sizes $H = W = 7, F_h = F_w = 3$. The ratio of L3 cache loads is given as $\frac{\texttt{im2col}}{\texttt{YaConv}}$, whereas the ratio for GFLOPS is $\frac{\texttt{YaConv}}{\texttt{im2col}}$. Values for both of these ratios in Figure 5.6 are presented with the corresponding color on the heatmap so that larger numbers are better for `YaConv`. The same color scales are used for both subfigures with each color scale adjusted to show an interval between the minimum and the maximum

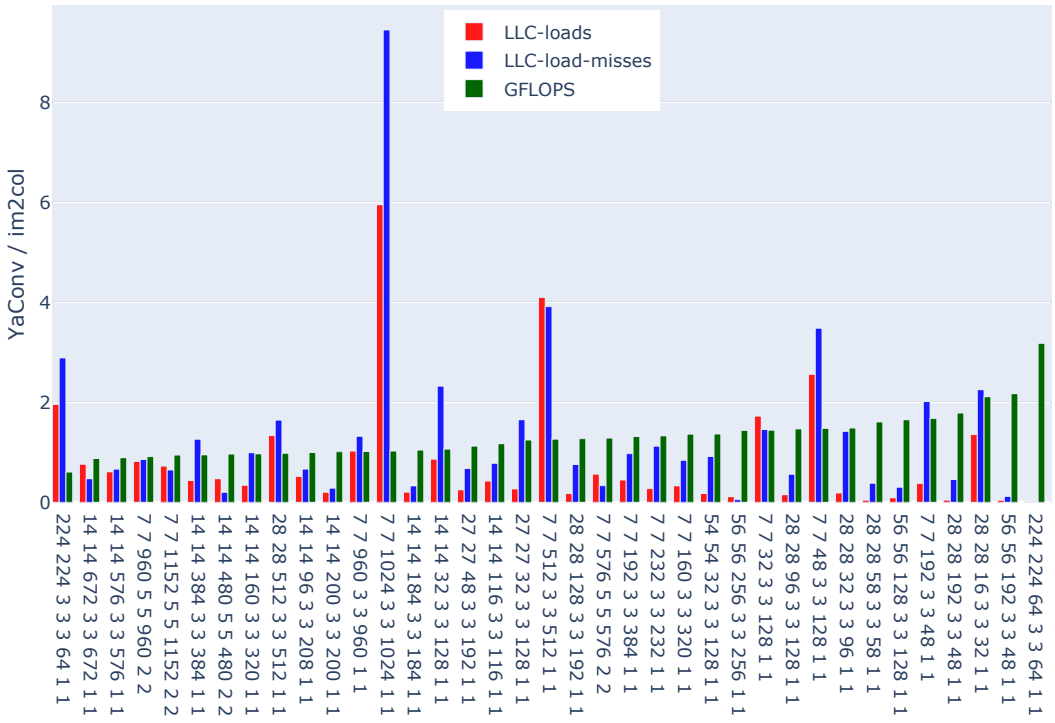
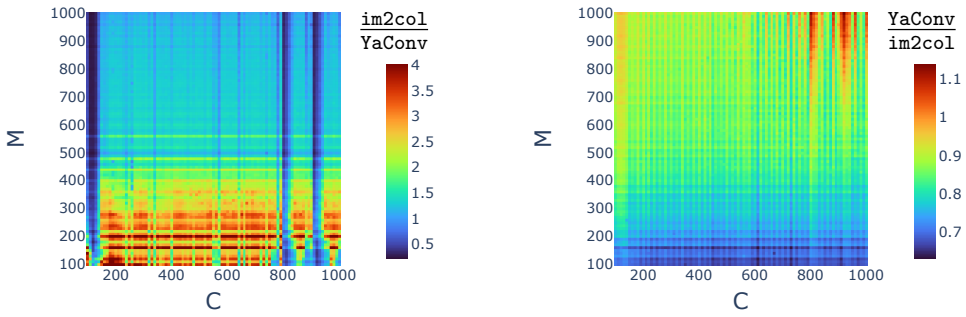


Figure 5.5: L3 cache usage and GFLOPS on PyTorch layers on Intel® Cascade Lake™

values on each heat map. Some locally large values in Figure 5.6a were clipped to 4 to prevent the figure from giving a distorted image of performance. The range of values was chosen to cover most of the poorly-performing layers from Figure 5.3.

When the number of output channels does not exceed the cache block size along that dimension ($M < 480$ in Figure 5.6a), the geometric mean of the reduction of L3 accesses of `YaConv` w.r.t `im2col` is 3×. With further increase in M, relative improvement of L3 usage in `YaConv` gets worse down to 1.6×. The heatmaps in Figures 5.6a and 5.6b indicate a correlation between the number of L3 cache loads and `YaConv` performance. Figure 5.6 exhibits a pattern of significantly worse L3 cache utilization by `YaConv` around specific values, *e.g.* $C < 60, C \approx 800, C \approx 940$. While performing experiments for a smaller range of parameters but with step size 1, we found that the same pattern persists on a finer scale and some input values cause a sudden increase in the whole memory hierarchy utilization. This could be explained by the parameter sizes



(a) L3 cache loads

(b) GFLOPS

Figure 5.6: Varying the number of input and output channels with fixed image and filter size on Intel® Cascade Lake™

(leading dimensions of tensors), coinciding with cache associativity stride and causing more hit conflicts than an average case. We observed similar trends on other machines and for the heat maps produced with other fixed image sizes.

5.4 Concluding Remarks

The main idea behind the design of `YaConv` is to prevent unnecessary copies of image elements, improve cache utilization, and make direct use of unmodified GEMM building blocks from high-performance numerical libraries. Achieving these goals required thinking of the input tensor as an $[W \cdot C] \times [H]$ matrix and packing it in the contiguous layout for outer-product GEMM matrices; packing the weight tensor in the same way that is done in the GEMM routine of the `im2col`-based convolution; calling the GEMM microkernel on these contiguously packed filter and image elements, and cleverly scattering the results of this computation into the resulting image. The resulting algorithm, `YaConv`, eliminates explicit image transformations, has a smaller memory footprint, and delivers superior performance than `im2col` by improving cache usage. `YaConv` is the first convolution algorithm that uses unmodified packing and GEMM microkernels from a numerical library to compute convolution without multiplying the number of input tensor elements kept in memory or the number of times these elements are loaded into the cache. A detailed performance study, varying parameters on multiple machines, helps understand

the superior performance of `YaConv` in comparison with `im2col`: it reduces the number of branches and LLC cache accesses.

Chapter 6

Related Work

6.1 A Brief History of Idiom Matching

David Callahan's seminal work identifies bounded-recurrence idioms in data-dependency graphs and replaces them with hand-written parallelized implementations in FORTRAN [14]. Another seminal work by Pinter et al. formalizes idiomatic recognition in a Computational Graph (CG) that is used to construct a tree where pattern matching is implemented [75]. They were among the first to normalize loops to improve idiom matching. Their normalization includes loop unrolling to guarantee that all loop-carried dependencies have a maximum distance of one. The goal is to improve idiom recognition by canonicalizing the code, thus eliminating syntactic variations that lead to subtle changes in the intermediate representation. To the best of our knowledge, there is no experimental evaluation of Pinter et al.'s work. KernelFaRer differs from Callahan's approach because it is integrated in the compilation, and differs from Pinter et al.'s approach because it does not build auxiliary structures for pattern matching, instead it relies only on the LLVM IR [64].

Menon et al. propose **Abstract Matrix Form (AMF)**, an algebraic language, as an IR and optimization language [68]. AMF expressions are transformed through a set of axioms used as rewrite rules provided by the user. AMF targets loop-based languages, such as FORTRAN and MATLAB, via source-to-source translation. The evaluation of AMF is restricted to simple matrix-matrix and matrix-vector kernels on a single platform. Birkbeck et al. also define a canonical representation for MATLAB programs and find idioms

to match patterns in an extensible database [8]. In contrast, KernelFaRer applies to any language once it is compiled to LLVM IR and is thoroughly evaluated across three platforms (see Section 4.2).

Sato Hiroyuki use grammar rewriting rules based on the array semantics of FORTRAN to propose a rule-based term rewritter for idiom recognition and replacement [44]. Later, Hiroyuki supports graph rewriting optimizations [45]. This approach is limited to FORTRAN, vulnerable to syntactic variations in the source code, and there is also no public evaluation of the approach.

He et al.'s tree-based idiom recognizer uses a **Reduced Affinity Relation Graph (RAG)**, a directed tree structure rooted at the left-hand side value of an assignment expression [41]. A RAG encodes data-dependency relations between values in an expression. Temporaries do not appear in a RAG. Likewise, KernelFaRer eliminates unnecessary temporaries through standard compiler transformation passes. He et al.'s approach incurs a memory and time overhead for creating and maintaining the RAG. In contrast KernelFaRer uses existing data representations already built during compilation.

Kawahito et al. match idioms to perform Just-In-Time compilation of JAVA string operations and replace them with IBM's System Z machine instructions [60]. This pattern matcher uses the **Abstract Syntax Tree (AST)** and the **Control-Flow Graph (CFG)**. Topological Embedding [27] enables them to implement non-exact idiom matching in the AST or CFG. Non-exact matches are made exact via a custom transformation pipeline. However, the transformation passes needed for non-exact matching limit the general applicability of this solution.

KernelFaRer is not the first work to recognize that customized implementations of idiomatic operations can match the performance of specialized libraries. Spampinato et al. describe SLinGEN, a DSL for expressing small-scale linear algebra applications [83]. SLinGEN optimizes an algebraic description of a kernel in its high-level representation and generates a C program with architecture-specific vector intrinsics. In contrast with KernelFaRer, SLinGEN was designed to target only small-scale and statically sized kernels. KernelFaRer is not limited to small, static kernels and automatically iden-

tifies the idioms in the code written in general purpose languages, no DSL specification required. Rink et al. present CFDlang, yet another DSL that targets tensor operations in the fluid dynamics application domain [80]. Similar to SLinGEN, CFDlang generates C programs that implement the idiom specified in its DSL and performs high-level optimizations as source-to-source transformations. KernelFaRer is not restricted to the fluid dynamics domain. In fact, KernelFaRer can readily identify GEMM and SYR2K in native programs without custom DSL specifications. The fact that KernelFaRer is integrated into LLVM, a major compiler framework, also sets it apart from both SLinGEN and CFDlang that are external tools.

The two most recent, closely related approaches to idiom matching are by Ginsbach et al. [34] and by Chelini et al. [18]. Ginsbach et al. introduce the **Idiom Description Language (IDL)**, a constraint-based DSL to describe idioms. IDL constraints are synthesized into LLVM IR passes that perform the matching process. While the first IDL prototype [34] had no automatic idiom-replacement mechanism, Ginsbach et al. later proposed a new DSL called LiLAC that includes automatic idiom replacement for both dense and sparse matrix-vector operations [33]. However, LiLAC's passes cannot identify the matrix layouts and the evaluation uses hard-coded calls with an assumed access order. Adoption, maintenance, and extension of IDL is challenging because it is external to the compilation framework. This prevents IDL users from exploiting rich debugging tools in LLVM and makes the task of specifying idioms much more involved since IDL is composed of several small components written in different languages. Similar limitations are found in Chelini et al.'s Loop Tactics [89], a DSL to describe idioms to be matched in a polyhedral schedule tree — an auxiliary structure constructed from LLVM IR, and to be replaced by either a library call or by code generated by Polly [37]. Adopting either of the DSL approaches is nontrivial. Loop Tactics requires understanding the polyhedral model and it is difficult to understand a complex composition of constraints in IDL. In contrast, KernelFaRer requires no auxiliary data structures and is fully based on standard LLVM IR passes, allowing developers experienced with LLVM's IR to extend its methodology to other idioms.

6.2 Compiler Approaches to Memory Access Optimization

In a seminal work, Goto and Van D. Geijn detail a layered approach to improve cache and vector-register utilization on CPUs [35]. Using this approach, modern linear algebra libraries, such as Eigen and BLAS, achieve high performance on HPC workloads. Goto and Van D. Geijn show that modelling both L2 cache and TLB — and not only L1 as considered earlier — is crucial for cache performance. That work is seminal because it publicly explained practical strategies for optimal cache and vector register utilization on CPUs; these strategies were previously only available in proprietary libraries. The layered strategy features two stages: 1. blocking input matrices and packing tiles of these blocks in such a way that tiles lay in main memory in the order that they will be accessed; and 2. computing a small GEMM at the register level. This paper is the first to create a compiler-only code generation for the layered approach and adapts blocking, tiling, and packing to create a data layout that is suitable for computing with MMA and to also improve utilization of the L3 cache.

Gareev et al. [28] implement tiling and packing within Polly [36] without the need for an external library or automatic tuning. Their approach with a hand-optimized SSE kernel reports performance on par with BLIS on Intel® Sandy Bridge. When not relying on an assembly kernel, their pass uses the default LLVM vectorizer that delivers only a small speedup over naive code. The solution proposed in this paper implements both memory optimization and micro kernel in the compiler, not requiring any hand-optimized code.

Uday Bondhugula presents an implementation of the BLAS strategy within the emerging MLIR framework [10]. He demonstrates that blocking, packing, register tiling, and unroll+jam yields code that is 34% slower than OpenBLAS on Intel®'s Coffee Lake [10]. Bondhugula also implemented a custom vectorization pass, to replace the default LLVM vectorizer, to achieve an additional 40% performance improvement, thus reaching 91% of the performance of OpenBLAS. Our experiments with Intel®, AMD® and IBM® Power9™ machines

also pointed out the weakness of the default LLVM vectorizer.

Carvalho et al. introduce KernelFaRer, a robust pattern recognition system that can identify matrix-multiplication patterns in the LLVM IR level and can replace these with library calls [16]. While this approach can lead to speedups on the order of 1000s in comparison with non-optimized code, it has the drawback of requiring the integration of libraries into a computer system that may not have it. Moreover, their experimental results indicate that, for smaller matrices, the overhead of invoking functions in the libraries leads to performance degradations. The solution in this paper is orthogonal to Carvalho et al.: their pattern recognition can identify GEMM kernels at the intermediate-level representation and then invoke the compiler-only solution presented here.

When presenting the ILLIAC IV, one of the first SIMD machines, Barnes et al. advocated that data parallelism would be crucial for progress [6], citing matrix operations as a critical target [62]. Nearly 50 years later, Barnes' direction culminated in the inclusion of vector extensions in all mainstream CPUs. Although fast vectorization is powerful, matrix-multiplication performance could be improved further with specialized hardware units. This possibility is now realized with the introduction of what Domke et al. have dubbed "matrix engines" [23].

Robust performance benchmarking is critical for the evaluation of vector extensions. While there is extensive performance evaluation of matrix multiplication on vector extensions for Intel architectures [2], [40], [42], to the best of our knowledge, similar studies do not exist for the PowerPC or Arm platforms. Moreover, the introduction of matrix engines is recent in all platforms and therefore only simulated or theorized performance estimates exist for AMX, SVE, or MMA [23], [76]. Therefore, this work is among the first to present performance evaluation of a matrix engine on actual hardware.

The advent of the "general purpose" GPUs quickly saw study and performance analysis of matrix computations [24], [63]. This evolved into implementations of matrix multiplications on GPUs: manually [65], through libraries like BLAS [69], and through frameworks such as DistME [39]. Matrix

multiplication is also central to the design of hardware for tensor-operation acceleration such as Google's Tensor Processing Unit [57], Nvidia's Tensor Core [67], and Huawei's Cube Unit [66].

Chapter 7

Conclusion

Utilizing matrix-multiplication building blocks efficiently in a software stack is essential for the generation of efficient code for high-performance computing and for deep learning applications. High-Performance Computing (HPC) applications often call specialized linear-algebra libraries directly, and these libraries have been tuned for the efficient computation of matrix multiplication in each architecture. In some cases, such as when the matrices are small, the overhead of a call to a library is not justified. Thus, there is a need for compiler solutions that use the building blocks efficiently.

This thesis presented two solutions for the efficient computation of matrix multiplication starting with user-level code that does not invoke linear-algebra libraries. The first, KernelFaRer, is based on an improved robust intermediate-level pattern recognition technique that can identify a matrix multiplication in the user code and replace it with a call to a library . The second, is an implementation of matrix multiplication in a production LLVM compiler that integrates the techniques from high-performance libraries. Experimental evaluation across several architectures and input sizes showed performance comparable to the target-specific library code. For the platforms and input cases when that was not the case, limitations of the vectorizing passes in LLVM were found. The proposed approach allowed to easily incorporate new matrix unit instructions into the existing code path.

For deep learning, an efficient implementation of the convolution operation is pivotal to neural network performance. Traditionally, convolution is computed

with a data transformation, followed by one or several calls to the matrix multiplication routine from a numerical library. This thesis presented a novel convolution algorithm design that does not rely on the library GEMM routine nor explicit input copy. The same memory hierarchy utilization principles that are used for matrix multiplication allowed the new algorithm to reduce the number of data accesses and narrow the gap between convolution and GEMM performance on CPUs. By carefully examining the data, we argued that this level of performance can be achieved for all convolution inputs by adjusting the input sizes to match architecture-dependent parameters. The algorithm performance benefits come at no memory cost which is a lucrative solution for embedded applications.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, principles, techniques*, 2nd ed. Addison wesley, 1986, ISBN: 0321486811. 13

[2] C. L. Alappat, J. Hofmann, G. Hager, H. Fehske, A. R. Bishop, and G. Wellein, "Understanding HPC benchmark performance on Intel Broadwell and Cascade Lake processors," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds., Frankfurt am Main, Germany, 2020, pp. 412–433, ISBN: 978-3-030-50743-5. 90

[3] AMD, *Software Optimization Guide for AMD Family 17th Models 30h and Greater Processors (Revision 3.01)*, 2020. 29

[4] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "High-Performance Low-Memory Lowering: GEMM-based Algorithms for DNN Convolution," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 99–106. DOI: 10.1109/SBAC-PAD49847.2020.00024. 68, 69

[5] *Arm® Architecture Reference Manual Armv8, for Armv8-A Architecture Profile*, Arm Limited, Jan. 2021. 39, 45, 65

[6] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV Computer," *IEEE Transactions on Computers*, vol. C-17, no. 8, pp. 746–757, 1968. DOI: 10.1109/TC.1968.229158. 90

[7] P. Bhat, J. Moreira, and S. K. Sadasivam, "Matrix-multiply assist best practices guide," IBM, Tech. Rep., 2021. [Online]. Available: https://www.redbooks.ibm.com/redpapers/pdfs/redp5612.pdf. 6

[8] N. Birkbeck, J. Levesque, and J. N. Amaral, "A Dimension Abstraction Approach to Vectorization in Matlab," in *International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, USA, 2007, pp. 115–130. 87

[9] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002. 9, 14, 26

[10] U. Bondhugula, *High Performance Code Generation in MLIR: An Early Case Study with GEMM*, 2020. arXiv: 2003.00532 [cs.PF]. [Online]. Available: https://arxiv.org/abs/2003.00532. 56, 59, 89

[11] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model," in *International Conference on Compiler Construction (ETAPS CC)*, Apr. 2008. 41

[12] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," in *Programming Language Design and Implementation*, Tucson, AZ, USA, 2008, 101–113. 9

[13] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-Generation Compute Benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, Berlin, Germany: Association for Computing Machinery, 2018, 41–42, ISBN: 9781450356299. DOI: 10.1145/3185768.3185771. [Online]. Available: https://doi.org/10.1145/3185768.3185771. 35

[14] D. Callahan, "Recognizing and parallelizing bounded recurrences," in *Languages and Compilers for Parallel Computing*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, 169–185, ISBN: 978-3-540-47063-2. 10, 11, 86

[15] J. P. L. de Carvalho, B. Kuzma, and G. Araujo, "Acceleration Opportunities in Linear Algebra Applications via Idiom Recognition," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20, Edmonton AB, Canada: Association for Computing Machinery, 2020, 34–35, ISBN: 9781450371094. DOI: 10.1145/3375555.3383586. [Online]. Available: https://doi.org/10.1145/3375555.3383586. 10

[16] J. P. L. de Carvalho, B. Kuzma, I. Korostelev, J. N. Amaral, C. Barton, J. Moreira, and G. Araujo, "KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls," *ACM Transactions On Architecture And Code Optimization (TACO)*, 2021. 55, 59, 90

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797. 35

[18] L. Chelini, O. Zinenko, T. Grosser, and H. Corporaal, "Declarative Loop Tactics for Domain-Specific Optimization," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Dec. 2019, ISSN: 1544-3566. DOI: 10.1145/3372266. [Online]. Available: https://doi.org/10.1145/3372266. 9, 11, 31, 88

[19] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*, G. Lorette, Ed., http://www.suvisoft.com, Université de Rennes 1, La Baule (France): Suvisoft, Oct. 2006. [Online]. Available: https://hal.inria.fr/inria-00112631. 14, 68

[20] M. Cho and D. Brand, "MEC: Memory-efficient convolution for deep neural network," in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 815–824. [Online]. Available: https://proceedings.mlr.press/v70/cho17a.html. 68, 69

[21] *Clang: a C language family frontend for LLVM*, https://clang.llvm.org, Accessed: January 2020. 15

[22] *cuBLAS — NVIDIA Developer*, https://developer.nvidia.com/cublas, Accessed: January 2020. 9

[23] J. Domke, E. Vatai, A. Drozd, P. Chen, Y. Oyama, L. Zhang, S. Salaria, D. Mukunoki, A. Podobas, M. Wahib, and S. Matsuoka, "Matrix engines for high performance computing:a paragon of performance or grasping at straws?" In *International Parallel and Distributed Processing Symposium*, 2021. 90

[24] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication," in *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS)*, Grenoble, France: Association for Computing Machinery, 2004, 133–137, ISBN: 3905673150. 90

[25] P Fischer and K Heisey, *NEKBONE: Thermal Hydraulics Mini-Application*, 2013. 35

[26] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," *Technical University of Denmark*, p. 383, Aug. 2019, Accessed: March 2020. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf. 29

[27] Fu, James Jianghai, "Directed Graph Pattern Matching and Topological Embedding," *Journal of Algorithms*, vol. 22, no. 2, pp. 372–391, 1997. 87

[28] R. Gareev, T. Grosser, and M. Kruse, "High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach," *ACM Transactions on Architecture And Code Optimization (TACO)*, vol. 15, no. 3, Sep. 2018, ISSN: 1544-3566. [Online]. Available: https://doi.org/10.1145/3235029. 59, 89

[29] R. van de Geijn and K. Goto, "Encyclopedia of parallel computing," in, D. Padua, Ed. Springer, 2011, ch. BLAS (Basic Linear Algebra Subprograms). [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_84. 6

[30] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 830–841. DOI: 10.1109/SC.2018.00069. 68

[31] E. Georganas, K. Banerjee, D. Kalamkar, S. Avancha, A. Venkat, M. Anderson, G. Henry, H. Pabst, and A. Heinecke, "Harnessing Deep Learning via a Single Building Block," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 222–233. DOI: 10.1109/IPDPS47924.2020.00032. 68

[32] E. Georganas, D. Kalamkar, S. Avancha, M. Adelman, C. Anderson, A. Breuer, J. Bruestle, N. Chaudhary, A. Kundu, D. Kutnick, F. Laub, V. Md, S. Misra, R. Mohanty, H. Pabst, B. Ziv, and A. Heinecke, "Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning Workloads," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21, St. Louis, Missouri: Association for Computing Machinery, 2021, ISBN: 9781450384421. DOI: 10.1145/3458817.3476206. [Online]. Available: https://doi.org/10.1145/3458817.3476206. 2

[33] P. Ginsbach, B. Collie, and M. F. P. O'Boyle, "Automatically Harnessing Sparse Acceleration," in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020, San Diego, CA, USA: Association for Computing Machinery, 2020, 179–190, ISBN: 9781450371209. DOI: 10.1145/3377555.3377893. [Online]. Available: https://doi.org/10.1145/3377555.3377893. 88

[34] P. Ginsbach, T. Remmelg, M. Steuwer, B. Bodin, C. Dubach, and M. F. P. O'Boyle, "Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, Williamsburg, VA, USA: Association for Computing Machinery, 2018, 139–153, ISBN: 9781450349116. DOI: 10.1145/3173162.3173182. [Online]. Available: https://doi.org/10.1145/3173162.3173182. 9, 11, 27–29, 88

[35] K. Goto and R. A.v. d. Geijn, "Anatomy of High-Performance Matrix Multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, May 2008, ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. [Online]. Available: https://doi.org/10.1145/1356052.1356053. 3, 4, 9, 14, 31, 38–40, 42

[36] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1 250 010, 2012. 58, 89

[37] T. C. Grosser, "Enabling polyhedral optimizations in LLVM," PhD thesis, Universität Passau, 2011. 9, 88

[38] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010. 7, 15, 26, 28, 38, 53, 55

[39] D. Han, Y.-M. Nam, J. Lee, K. Park, H. Kim, and M.-S. Kim, "DistME: A fast and elastic distributed matrix computation engine using GPUs," in *International Conference on Management of Data (SIGMOD)*, Amsterdam, Netherlands, 2019, 759–774, ISBN: 9781450356435. [Online]. Available: https://doi.org/10.1145/3299869.3319865. 90

[40] S. A. Hassana, A. M. Hemeida, and M. M. M. Mahmoud, "Performance Evaluation of Matrix-Matrix Multiplications Using Intel's Advanced Vector Extensions (AVX)," *Microprocessors and Microsystems*, vol. 47, pp. 369–374, 2016, ISSN: 0141-9331. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933116302502. 90

[41] J. He, A. E. Snavely, R. F. Van der Wijngaart, and M. A. Frumkin, "Automatic Recognition of Performance Idioms in Scientific Applications," in *2011 IEEE International Parallel & Distributed Processing Symposium*, IEEE, 2011, pp. 118–127. 11, 87

[42] A. Hemeida, S. Hassan, S. Alkhalaf, M. Mahmoud, M. Saber, A. M. Bahaa Eldin, T. Senjyu, and A. H. Alayed, "Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor," *Ain Shams Engineering Journal*, vol. 11, no. 4, pp. 1179–1190, 2020, ISSN: 2090-4479. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2090447920300058. 90

[43] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006. 35

[44] S. Hiroyuki, "Array Form Representation of Idiom Recognition System for Numerical Programs," *SIGAPL APL Quote Quad*, vol. 31, no. 2, 87–98, Dec. 2000, ISSN: 0163-6006. DOI: 10.1145/570406.570418. [Online]. Available: https://doi.org/10.1145/570406.570418. 11, 87

[45] Hiroyuki, Sato, "Idiom Recognition and Program Scheme Recognition Based Program Transformations for Performance Tuning–Beyond Compiler Optimizations–," in *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, IEEE, 2009, pp. 272–279. 11, 87

[46] R. A. Horn and C. R. Johnson, *Matrix analysis*. Cambridge university press, 2012. 13

[47] J. Huang and R. A. Van de Geijn, "BLISlab: A sandbox for optimizing GEMM," *arXiv preprint arXiv:1609.00076*, 2016. 32

[48] IBM, *Power9 Processor User's Manual (Version 2.0)*, 2018. 29

[49] ——, *ESSL Guide and Reference (Version 5, Release 5)*, 2020. 14, 26, 28

[50] IBM, "Power ISA version 3.1," IBM, Tech. Rep., 2020. [Online]. Available: `https://ibm.ent.box.com/s/hhjfw0x0lrbtyzmiaffnbxh2fuo0fog0`. 5

[51] Intel, *Intel math kernel library: Developer reference manual (revision 26)*, 2020. 14, 26, 28

[52] Intel Corporation, *Oneapi deep neural network library (onednn)*, `https://github.com/oneapi-src/oneDNN`, 2016. 68

[53] *Intel® Architecture Instruction Set Extensions and Future Features Programming Reference*, Intel Corporation, Feb. 2021. 7, 39, 40, 65, 66

[54] K. E. Iverson, "A Programming Language," in *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, ACM, 1962, pp. 345–351. 11, 12

[55] Y. Jia, PhD thesis, 2014, pp. 74–76. [Online]. Available: `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-93.pdf`. 1

[56] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014. 78

[57] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *International Symposium on Computer Architecture (ISCA)*, Toronto, ON, Canada, 2017, 1–12, ISBN: 9781450348928. [Online]. Available: `https://doi.org/10.1145/3079856.3080246`. 91

[58] P. S. Juan, A. Castelló, M. F. Dolz, P. Alonso-Jordá, and E. S. Quintana-Ortí, "High Performance and Portable Convolution Operators for ARM-based Multicore Processors," *ArXiv*, vol. abs/2005.06410, 2020. 68, 69

[59] T. Kalibera and R. Jones, "Rigorous Benchmarking in Reasonable Time," in *Proceedings of the 2013 International Symposium on Memory Management*, ser. ISMM '13, Seattle, Washington, USA: Association for Computing Machinery, 2013, 63–74, ISBN: 9781450321006. DOI: 10.1145/2491894.2464160. [Online]. Available: https://doi.org/10.1145/2491894.2464160. 30

[60] M. Kawahito, H. Komatsu, T. Moriyama, H. Inoue, and T. Nakatani, "Idiom Recognition Framework Using Topological Embedding," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 3, Sep. 2013, ISSN: 1544-3566. DOI: 10.1145/2512431. [Online]. Available: https://doi.org/10.1145/2512431. 11, 87

[61] D. Khaldi, Y. Luo, B. Yu, A. Sotkin, B. Morais, and M. Girkar, "Extending LLVM IR for DPC++ Matrix Support: A Case Study with Intel® Advanced Matrix Extensions (Intel® AMX)," in *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2021, pp. 20–26. DOI: 10.1109/LLVMHPC54804.2021.00008. 1

[62] D. J. Kuck, "ILLIAC IV Software and Application Programming," *IEEE Transactions on Computers*, vol. C-17, no. 8, pp. 758–770, 1968. DOI: 10.1109/TC.1968.229159. 90

[63] E. S. Larsen and D. McAllister, "Fast Matrix Multiplies Using Graphics Hardware," in *ACM/IEEE Conference on Supercomputing*, ser. SC '01, Denver, Colorado, 2001, p. 55, ISBN: 158113293X. [Online]. Available: https://doi.org/10.1145/582034.582089. 90

[64] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04, Palo Alto, California: IEEE Computer Society, 2004, pp. 75–, ISBN: 0-7695-2102-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673. 10, 12, 20, 86

[65] J. Li, S. Ranka, and S. Sahni, "Strassen's Matrix Multiplication on GPUs," in *International Conference on Parallel and Distributed Systems (ICPADS)*, Tainan, Taiwan, 2011, pp. 157–164. 90

[66] H. Liao, J. Tu, J. Xia, and X. Zhou, "Davinci: A scalable architecture for neural network computing," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, IEEE Computer Society, 2019, pp. 1–44. 91

[67] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA Tensor Core Programmability, Performance Precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091. 91

[68] V. Menon and K. Pingali, "High-Level Semantic Optimization of Numerical Codes," in *Proceedings of the 13th International Conference on Supercomputing*, ser. ICS '99, Rhodes, Greece: Association for Computing Machinery, 1999, 434–443, ISBN: 158113164X. DOI: 10.1145/305138.305230. [Online]. Available: https://doi.org/10.1145/305138.305230. 10, 11, 86

[69] R. Nath, S. Tomov, and J. Dongarra, "Accelerating GPU kernels for dense linear algebra," in *High Performance Computing for Computational Science – VECPAR*, J. M.L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds., Berkeley, CA, USA, 2010, pp. 83–92, ISBN: 978-3-642-19328-6. 90

[70] *OpenBLAS: An optimized BLAS library*, https://www.openblas.net/, Accessed: January 2020. 3, 4, 14, 28, 31, 68, 69, 7

[71] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1201, 1986. 21, 24

[72] J. Palsberg and C. B. Jay, "The essence of the visitor pattern," in *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241)*, IEEE, 1998, pp. 9–15. 11

[73] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf. 78

[74] Perlis, Alan J. and Rugaber, Spencer, "Programming with Idioms in APL," *SIGAPL APL Quote Quad*, vol. 9, no. 4-P1, 232–235, May 1979, ISSN: 0163-6006. DOI: 10.1145/390009.804466. [Online]. Available: https://doi.org/10.1145/390009.804466. 9–11

[75] S. S. Pinter and R. Y. Pinter, "Program Optimization and Parallelization Using Idioms," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, 305–327, May 1994, ISSN: 0164-0925. DOI: 10.1145/177492.177494. [Online]. Available: https://doi.org/10.1145/177492.177494. 10, 11, 86

[76] A. Poenaru and S. McIntosh-Smith, "Evaluating the effectiveness of a vector-length-agnostic instruction set," in *Euro-Par: International European Conference on Parallel and Distributed Computing*, M. Malawski and K. Rzadca, Eds., Warsaw, Poland, 2020, pp. 98–114, ISBN: 978-3-030-57675-2. 90

[77] L.-N. Pouchet and T. Yuki, *PolyBench/C 4.2.1: The polyhedral benchmark suite*, http://polybench.sf.net, 2019. 24, 29

[78] *Power ISA Version 3.1*, IBM Corporation, May 2020. 39

[79] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70. DOI: 10.1109/HPCA47549.2020.00015. 38

[80] N. A. Rink, I. Huismann, A. Susungi, J. Castrillon, J. Stiller, J. Fröhlich, and C. Tadonki, "CFDlang: High-level code generation for high-order methods in fluid dynamics," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, 2018, pp. 1–10. 88

[81] T. M. Smith, R. Van De Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of High-Performance Many-Threaded Matrix Multiplication," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IEEE, 2014, pp. 1049–1059. 31

[82] Snyder, Lawrence, "Recognition and selection of idioms for code optimization," *Acta Informatica*, vol. 17, no. 3, 327–348, 1982, ISSN: 1432-0525. DOI: {10.1007/BF00264357}. [Online]. Available: {https://doi.org/10.1007/BF00264357}. 11

[83] D. G. Spampinato, D. Fabregat-Traver, P. Bientinesi, and M. Püschel, "Program generation for small-scale linear algebra applications," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 327–339. 87

[84] W. J. Starke, B. W. Thompto, J. A. Stuecheli, and J. E. Moreira, "IBM's POWER10 Processor," *IEEE Micro*, vol. 41, no. 2, pp. 7–14, 2021. DOI: 10.1109/MM.2021.3058632. 1, 39

[85] S. Tavarageri, A. Heinecke, S. Avancha, B. Kaul, G. Goyal, and R. Upadrasta, "PolyDL: Polyhedral Optimizations for Creation of High-Performance DL Primitives," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, 2021, ISSN: 1544-3566. DOI: 10.1145/3433103. [Online]. Available: https://doi.org/10.1145/3433103. 68

[86] *The Science of High-Performance Computing Group*, https://shpc.oden.utexas.edu/, Accessed: March 2020. 32

[87] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating blas functionality," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, Jun. 2015, ISSN: 0098-3500. [Online]. Available: https://doi.org/10.1145/2764454. 38

[88] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 2017, pp. 19–24. 38

[89] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, "Schedule Trees," in *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria*, 2014. 88

[90] H. Waugh and S. McIntosh-Smith, "On the Use of BLAS Libraries in Modern Scientific Codes at Scale," in *Smoky Mountains Computational Sciences and Engineering Conference*, Springer, 2020, pp. 67–79. 38, 39

[91] V. M. Weaver, *Linux perf event Features and Overhead*, Austin, TX: University of Maine, 2013. 78

[92] M. Weißbrich, A. García-Ortiz, and G. Payá-Vayá, "Comparing vertical and horizontal SIMD vector processor architectures for accelerated image feature extraction," *Journal of Systems Architecture*, vol. 100, p. 101 647, 2019, ISSN: 1383-7621. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S1383762119304540`. 7

[93] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, 2012, pp. 684–691. DOI: `10.1109/ICPADS.2012.97`. 7, 9, 26, 28, 38, 53, 55

[94] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, 14:1–14:33, Jun. 2015. [Online]. Available: `http://doi.acm.org/10.1145/2764454`. 3, 4, 14, 26, 28, 31, 68, 6