Region-Based Data Layout Transformations: Compiler Techniques and Applications

by

Caio Salvador Rohwedder

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

Doctor of Philosophy

Department of Computing Science

University of Alberta

© Caio Salvador Rohwedder, 2025

Abstract

Region-based data layout transformations are compiler optimizations that restructure data layouts via data copying. They are named region-based because they only affect specific code regions, such as loops. These transformations improve cache and translation look-aside buffer (TLB) locality, allow better compiler code generation, and work synergistically with loop transformations.

This thesis begins by exploring why region-based data layout transformations are not present in mainstream compilers, unlike loop transformations such as tiling. It focuses on data packing, an essential optimization in high-performance General Matrix-Matrix Multiplication (GEMM), which remains largely unsupported in compilers. The first contribution is GPAT, a Generalized Packing Analysis and Transformation that applies packing, when beneficial, to a generic input loop nest. Implemented in the Affine dialect of MLIR and evaluated on the PolyBench/C suite, GPAT generalizes packing beyond GEMM, obtaining significant speedups over state-of-the-art loop optimizers like Pluto and Polly.

While GPAT targets dense tensor computations, the second contribution, RebaseDL, extends its core ideas to data structures. Structure splitting and field reordering typically reorganize data structures globally and often depend on profiling to guide their transformations. RebaseDL is a static analysis that identifies profitable region-based structure splitting and field reordering based on data reuse, enabling improved locality without global restructuring. Implemented in LLVM, RebaseDL detects multiple transformation opportunities in the SPEC CPU benchmark suite, obtaining up to 1.34x speedups in targeted regions.

The third contribution introduces ZConv, a novel algorithm for convolutional layers of convolutional neural networks, which demonstrates an indirect application of packing. A common approach to optimize convolution is to reduce the operation to GEMM using Image-to-Column (Im2col), which may require data copying and data duplication. ZConv is a new formulation that reduces convolution to GEMM for CPU architectures using zero copies, by giving non-contiguous matrix inputs to GEMM. Data packing employed in GEMM allows high-performance execution even in the presence of such inputs. When integrated into PyTorch, ZConv yields up to 1.8x speedups in end-to-end vision model inference.

Preface

Chapter 2 of this thesis has been published as **Caio Salvador Rohwedder**, Nathan Henderson, João P. L. De Carvalho, Yufei Chen, and José Nelson Amaral, "To Pack or Not to Pack: A Generalized Packing Analysis and Transformation", Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23). This work was presented by the author at the CGO conference in Montreal, Canada, on February 2023, where it won a Distinguished Paper Award. José Nelson Amaral and João P. L. De Carvalho were supervising authors, guiding the research direction, helping define good practices for the experimental evaluation, and reviewing and editing the manuscript. Yufei Chen contributed benchmarks for the evaluation. I was responsible for the initial ideas that led to this research, and Nathan Henderson and I collaborated on writing the manuscript, designing (and implementing) the analysis and transformation, and performing the evaluation.

Chapter 3 of this thesis has been published as **Caio Salvador Rohwedder**, João P. L. De Carvalho, and José Nelson Amaral, "Region-Based Data Layout via Data Reuse Analysis", Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC 2024). This work was presented by the author at the CC conference in Edinburgh, United Kingdom, in March 2024. José Nelson Amaral and João P. L. De Carvalho served as supervising authors, guiding the research and reviewing the manuscript. I was responsible for designing the ideas presented in the paper, implementing them, running the evaluation, and writing the manuscript.

Chapter 4 of this thesis has been submitted for publication as **Caio Salvador Rohwedder**, João Paulo Labegalini De Carvalho, and José Nelson Amaral, "Zero-Copy GEMM-Based Fast Convolution". José Nelson Amaral and João Paulo Labegalini de Carvalho were supervisory authors, guiding how to present and evaluate this work and reviewing the manuscript. I identified the opportunity for this work, designed

its algorithm, performed its evaluation, and wrote the manuscript.

To Jessica

For your love and support through this journey.

Acknowledgements

I would like to thank my supervisor Dr. José Nelson Amaral for his invaluable guidance and support, and for inviting me to begin this Ph.D. at the University of Alberta. I am also grateful to my co-supervisor, Dr. João Paulo Labegalini de Carvalho, for always helping me along the way, even before the start of this degree. It has been a privilege to work with both of you.

In addition, I extend my thanks to all the incredible people at the Systems Laboratory that researched and collaborated with me, and that also simply were there for insightful discussions and friendly conversation. A special thanks to Nathan Henderson for your friendship and for collaborating with me on the first project of this thesis.

This research is partially supported by funding from the Natural Sciences and Engineering Research Council of Canada and by an academic fellowship sponsored by Huawei Technologies Co. Ltd.

Contents

1 Introduction **1**

2 To Pack or Not to Pack: A Generalized Packing Analysis and Transformation **4**
2.1 Background . . . 6
2.2 Packing Analysis . . . 8
2.2.1 Preliminary Definitions and Notation . . . 8
2.2.2 Analysis Constraints . . . 9
2.2.3 Running Example . . . 9
2.2.4 Analysis Overview . . . 10
2.2.5 GPAT in the MLIR Affine Dialect . . . 16
2.3 GPAT Evaluation . . . 17
2.3.1 Setup . . . 17
2.3.2 Experimental Methodology . . . 18
2.3.3 Packing Selection Evaluation . . . 19
2.3.4 PolyBench Evaluation . . . 23
2.4 Discussion . . . 26

3 Region-Based Data Layout via Data Reuse Analysis **28**
3.1 Definitions and Notation . . . 30
3.2 Data Layout Transformation . . . 31
3.3 RebaseDL: The Analysis . . . 33
3.3.1 Analysis Phases . . . 34
3.3.2 Underneath the Analysis . . . 41
3.3.3 RebaseDL in LLVM . . . 42
3.4 RebaseDL Evaluation . . . 43
3.4.1 Setup . . . 43
3.4.2 Experimental Methodology . . . 45
3.4.3 Data Reuse Impact . . . 45
3.4.4 Transformation Evaluation . . . 46
3.5 Discussion . . . 50

4 Zero-Copy GEMM-Based Fast Convolution **52**
4.1 Convolution and GEMM . . . 54
4.2 Zero-Copy Convolution . . . 59
4.2.1 ZConv Algorithm . . . 59
4.2.2 ZConv Algorithm Support for Dilation and Grouped Convolution 63
4.2.3 Integration of ZConv with PyTorch . . . 65
4.3 ZConv Evaluation . . . 66
4.3.1 Setup . . . 66
4.3.2 Standalone Convolutions . . . 67
4.3.3 End-to-end Models . . . 76
4.4 Discussion . . . 80

5 Related Work **82**
5.1 Data Packing Optimizations 82
5.2 Data Structure Splicing 84
5.3 GEMM-Based Convolutional Layers 85

6 Conclusion **89**

References **91**

List of Tables

3.1 Experimental machine configuration 44
3.2 The number of candidates found by RebaseDL in Standard Performance Evaluation Corporation (SPEC) Central Processing Unit (CPU) benchmarks. In total, 71 candidates were found. 44
3.3 Execution time of Clang and RebaseDL for code regions of SPEC benchmarks not shown in Figure 3.4. The ± values represent the 95% confidence interval around the mean execution times reported. 48
3.4 Speedup and reuse iterations for code regions of SPEC benchmarks not shown in Figure 3.4. 49

4.1 Notation for convolution parameters and input tensor dimensions. . . 55
4.2 Experimental configuration 66
4.3 Software versions 66
4.4 Models and convolution layers per source 66
4.5 Types of layers in the models of Table 4.4. Types are not mutually exclusive. 66
4.6 Tensor layouts used by each method in Section 4.3.2. 67
4.7 Parameters for convolution layers used in comparisons of memory hierarchy metrics 69
4.8 Occurrences of convolution layers used in comparisons of memory hierarchy metrics 70
4.9 Memory hierarchy metrics of **Yaconv / ZConv using BLIS** as BLAS 70
4.10 Memory hierarchy metrics of **Im2col / ZConv**. Single-thread execution 73
4.11 Memory hierarchy metrics of **LibTorch / LibTorch-ZConv**. Single-thread execution 75
4.12 Total execution time of running inference on all images in a validation set of the Pascal VOC 2012 dataset with *DeepLabV3+* models using PyTorch and PyTorch-ZConv. 80

List of Figures

2.1 Packing selection evaluation graphs 20
2.2 *2mm* L1 Data Translation Lookaside Buffer (dTLB) load misses. . . . 23
2.3 *2mm* instructions count. 23
2.4 Polybench evaluation on the Polymer tiling engine 24
2.5 Polybench evaluation on the Affine tiling engine. Baseline times (ms) of each benchmark in the same order of figure: 1673, 3121, 482, 93, 13919, and 2966 25

3.1 Type `f1_neuron` is split into two types, one containing only field `P`, and another omitted from the figure. The cache lines show the disposition of field `P` in an array of these structures. 29
3.2 Control-Flow Graph (CFG) of a loop nest and its dominator tree. . . 39
3.3 Speedup of RebaseDL over Clang while varying the number of reuse iterations for the region *trainmatch* of *179.art*. 46
3.4 Execution time before and after the transformation for code regions in the benchmarks *175.vpr*, *179.art*, and *445.gobmk*. From left to right, RebaseDL obtains the following speedups over Clang: 1.03x, 1.24x, 1.34x, 1.05x. 47

4.1 View of a naive convolution that uses unit stride, no padding, no dilation, and a 2×2 filter. The tensors have only height and width dimensions, the image is 4×4, and the output is 3×3. Black arrows indicate loop iterations, and colors indicate the data flow. The operation at the center is an element-wise multiplication followed by a sum reduction. 57
4.2 On the left, the input tensors to the convolution and their respective dimensions (omitting the batch dimension). On the right, an example of a General Matrix-Matrix Multiplication (GEMM) within ZConv, highlighting its input elements and dimensions. 57
4.3 View of ZConv in a simplified convolution that uses unit stride, no padding, no dilation, and a 2x2 filter. The tensors have only height and width dimensions, the image is 4×4, and the output is 3×3. Black arrows indicate loop iterations, and colors indicate the data flow. The operation at the center is a GEMM, its output is accumulated into the output. 62
4.4 Improvement of **ZConv over Yaconv** on individual *Conv2D* layers. Single-thread execution. The graphs exclude layers that are pointwise, unsupported by Yaconv, and for which Yaconv generated incorrect results. The graph showing ZConv using BLIS is included because Yaconv is integrated into BLIS. 71
4.5 Improvement of **ZConv over Im2col** on individual *Conv2D* layers. Both graphs exclude pointwise layers, and single-thread execution also excludes grouped layers. 72

4.6 Improvement of **LibTorch-ZConv over LibTorch** on individual *Conv2D* layers. Both graphs exclude pointwise grouped layers. Additionally, they exclude layers where output height or output width is greater than or equal to input channels. 74
4.7 Improvement of **PyTorch-ZConv over PyTorch** on end-to-end models with single-thread execution. 78
4.8 Improvement of **PyTorch-ZConv over PyTorch** on end-to-end models with multi-thread execution. 79

Glossary

Application Programming Interface (API)

Artificial Intelligence (AI)

Basic Linear Algebra Subroutines (BLAS)

Central Processing Unit (CPU)

Control-Flow Graph (CFG)

Convolutional Neural Network (CNN)

Data Translation Lookaside Buffer (dTLB)

Deep Learning (DL)

Fast Fourier Transform (FFT)

Feedback-Directed Optimization (FDO)

General Matrix-Vector Multiplication (GEMV)

General Matrix-Matrix Multiplication (GEMM)

Graphics Processing Unit (GPU)

Induction Variable (IV)

Instruction Set Architecture (ISA)

Intermediate Representation (IR)

Just-In-Time (JIT)

Least Recently Used (LRU)

Red Green Blue (RGB)

Scalar Evolution (SCEV)

Single-Entry Single-Exit (SESE)

Standard Performance Evaluation Corporation (SPEC)

Static Single-Assignment (SSA)

Static Control Parts (SCoPs)

Translation Lookaside Buffer (TLB)

Very Long Instruction Word (VLIW)

Chapter 1

Introduction

This thesis states that both compile-time and runtime region-based data layout transformations can improve the performance of code regions by increasing data locality. This statement is supported by contributions where the compiler leverages the order of load/store operations issued to orchestrate the data movement through the memory hierarchy, or where a compile-time reuse analysis increases the efficiency of region-based data-layout transformations.

Motivated by the success of such transformations in high-performance GEMM, this work begins with a compiler approach that is based on automatically identifying code regions with data reuse. This direction leads to the first two contributions of the thesis: GPAT and RebaseDL. Chapter 2 presents GPAT, a compiler analysis and transformation that applies packing to generic loop nests involving dense tensor computations. Chapter 3 builds upon these ideas to address more general-purpose code, introducing RebaseDL, an analysis for region-based structure splitting and field reordering. Finally, this thesis explores runtime data layout transformations. By leveraging the runtime data layout transformations present in GEMM due to its intrinsic data reuse, a new algorithm is proposed that composes GEMM operations using non-contiguous matrix inputs. The third contribution, described in Chapter 4, is a new high-performance GEMM-based algorithm for convolutional layers, a core component of Convolutional Neural Networks (CNNs).

GEMM is a numerical computation of critical importance in both scientific applications and Deep Learning (DL) workloads. GEMM has long been optimized to near-peak performance on CPUs, and many other matrix-matrix computations

are efficiently implemented based on it [34]. To achieve such performance, its implementation is typically divided into macro and micro kernels [33], [81]. The micro kernel performs matrix multiplications that produce a small, fixed-size output, and it is often hand-optimized in assembly to maximize performance [9]. The macro kernel is responsible for decomposing the GEMM operation into smaller computations until they match the micro kernel size. This decomposition is achieved through loop tiling, and loop interchange determines the order in which the matrices are partitioned. The macro kernel is carefully structured to exploit the natural data reuse patterns of GEMM and the memory hierarchy of modern CPUs. However, to reach peak performance, the micro kernel may require its input submatrices, which are produced by tiling, to follow a memory layout that is not present in the original matrices. This layout is created through data packing, a process that copies and reorganizes the matrix data during the GEMM computation to match the expected layout and to further optimize the computation for the memory hierarchy. GPAT (Chapter 2) extracts the packing strategies present in high-performance GEMM implementations into a compiler analysis and transformation, allowing packing to be applied to generic loop nests that access dense tensors. Compared to loop optimizers such as LLVM's Polly [35], GPAT achieves up to a 4× speedup in the PolyBench/C [73] benchmark suite.

In the domain of general-purpose compilers, such as LLVM [50], data layout transformation techniques like data structure splitting and field reordering have been widely studied. These techniques reorganize how aggregate data structures are laid out in memory to improve cache locality throughout program execution [15], [92]. They are typically applied by first profiling the program to determine how the structure fields should be reorganized. Then, this reordering strategy is applied globally at allocation time, instructing the dynamic memory allocator to lay out structures in a predefined order. This general approach has two main problems. First, profiling incurs significant runtime overhead to generate an allocation policy. Second, applying a single global memory layout can only approximate an ideal layout for the program as a whole; therefore, code regions that access data structures differently may incur performance penalties. A region-based alternative, similar to packing, was previously proposed by Temam *et al.* [79], but it overlooked a key optimization

component: data reuse. RebaseDL (Chapter 3) introduces a region-based structure splitting and field reordering analysis that identifies data reuse opportunities and addresses both limitations of prior approaches. In experiments on the SPEC CPU benchmark suite, RebaseDL identifies many opportunities for region-based data layout transformations, achieving up to a 1.34× speedup in transformed regions.

While GPAT and RebaseDL are inspired by the use of packing in high-performance GEMM, ZConv (Chapter 4) takes the opposite approach: it leverages GEMM to optimize convolutional layers, a key component in CNNs. GEMM-based algorithms are a common strategy for accelerating convolutional layers; they work by reusing highly optimized GEMM kernels from linear algebra libraries, typically through the Im2col transformation. Im2col is a data layout transformation that copies and may duplicate input image data into a new matrix format [10]. This new matrix is used as an input so that convolution is executed as GEMM. Although effective, this transformation introduces overhead due to the extra memory usage and data movement. Recent approaches have aimed to reduce the cost of Im2col [2], [77]. Some approaches sacrifice portability in favor of custom GEMM implementations, while others retain unmodified GEMM kernels by splitting the convolution into smaller GEMM operations, at the cost of decreased data locality. However, previous works fail to fully eliminate Im2col and other data copies while using unmodified GEMM kernels. ZConv introduces a novel convolution algorithm that entirely removes the need for data copying while still using unmodified GEMM kernels. Its core insight is that GEMM supports non-contiguous input matrices via configurable stride and size parameters. Despite non-contiguous inputs, the internal packing transformations performed by GEMM kernels allow the computation to remain highly efficient. As a result, integrating ZConv into PyTorch yields speedups of up to 1.8× in end-to-end vision model inference.

The remaining chapters of this thesis are organized as follows. Chapter 2 describes GPAT, Chapter 3 describes RebaseDL, and Chapter 4 describes ZConv. Finally, Chapter 5 reviews related work, and Chapter 6 concludes the thesis.

Chapter 2

To Pack or Not to Pack: A Generalized Packing Analysis and Transformation

In large part, the success of numerical computations is due to the development of very efficient linear-algebra libraries [36], [81], [88]. A sophisticated implementation of a GEMM kernel that relies upon two transformations – tiling and packing – is key to libraries' efficiency [33], [34]. Tiling breaks large matrices into smaller blocks that fit in each level of the memory hierarchy. Packing creates a copy of such blocks while reorganizing their elements in the order in which each element is used by computations. Packing leads to higher spatial locality and higher cache hit rates. Higher spacial locality increases the likelihood that consecutively accessed data belongs to the same memory page. Therefore, packing also reduces the number of Translation Lookaside Buffer (TLB) entries required to address elements within a given loop-nest level.

While loop tiling [68] is widely available as a transformation pass in loop optimizers, packing is almost exclusively used in handcrafted matrix-multiplication implementations. The absence of packing as a general compiler transformation is likely because, until now, the requirements to identify a profitable packing opportunity for general computations had not been defined. For instance, a modular packing pass based on principled cost-benefit analysis for general computations is absent from loop transformations in TVM [14], Halide [74], the Affine dialect of MLIR [52], and Polyhedral frameworks (*e.g.* Pluto [6] and Polly [35]), and optimizing compilers

in general. TVM and the MLIR Affine dialect support explicit memory transfers designed for software-controlled memory buffers found in Graphics Processing Units (GPUs) and other accelerators. Bondhugula demonstrates that for such transfer operations to emulate packing, one must follow a pre-defined strategy that does not generalize [5]. Polly may apply packing when it successfully pattern matches a computation to determine that it is a matrix multiplication but it cannot apply packing to a general computation. Moreover, Polly's pattern matching is limited to specific ways of writing the matrix multiplication code [22]. Wu *et al.* [87] can apply packing via a pragma directive added to C code. Such pragma must be added manually. Overall, the use of packing is restricted to the handcrafting of specific computations such as tiled matrix multiplication in predefined loop orderings or auto-tuning techniques.

There are two central ideas in this chapter. First, a compiler framework can implement a modular packing strategy that applies packing to a generic loop nest, thus extending packing beyond handcrafted matrix-multiplications. Second, an analytical-model approach to determine *what* and *where* to pack is a viable alternative to trial-and-error auto-tuning of packing. This chapter proposes **Generalized Packing Analysis and Transformation** (GPAT) based on these ideas. GPAT's analysis is inspired by the use of packing described by Goto *et al.* [33] to optimize GEMM and GPAT's code transformation builds upon the infrastructure created by Bondhugula [5] to experiment with packing in GEMM.

The main contributions of this chapter are the following:

- GPAT: a modular compile-time packing analysis and transformation that uses an analytical model to decide when packing reduces TLB misses and enables vectorization through data-layout changes (Section 2.2)
- An open-source artifact that implements GPAT in the MLIR Affine dialect (Section 2.2.5)
- The evaluation of GPAT. It shows that GPAT's heuristic selects good combinations of packings, achieving significant speedup over the Polly [35] and Pluto [6] loop optimizers in the Polybench [73] benchmark suite. Additionally, the evaluation demonstrates that GPAT is orthogonal to tiling; it can apply packing and improve the performance of loop nests with or without a prior tiling transformation and it

does so when a loop nest is tiled by two different approaches at multiple tiling factors (Section 2.3).

2.1 Background

Many computations access data that is not contiguous in memory, such as a submatrix within a larger matrix, leading to poor spatial locality. Besides poor cache performance, such an access pattern may stress the virtual memory system by requiring that virtual addresses belonging to many virtual pages be translated within a single loop nest. Address translation is made efficient through the use of a TLB, which is a cache for address translations. However, the limited number of TLB entries may not be enough to translate all the addresses accessed in a loop nest. Therefore it is crucial to generate code that either avoids or reduces the number of cache and TLB misses to obtain good performance. Packing is a technique that can increase cache and TLB utilization.

Packing consists of making an in-memory copy of a *subtensor* from an n-dimensional tensor. A subtensor is a block of tensor elements, typically non-contiguous in memory. During the packing copy, the order of subtensor elements may be rearranged to make consecutively accessed elements contiguous in memory. Packing a non-contiguous subtensor results in three important benefits. First, fewer cache self-interference misses (commonly referred to as *conflict misses*) occur when accessing the packed subtensor because contiguous elements are less likely mapped to the same cache set [49]. This benefit is the main motivation in earlier works that proposed packing because, at that time, caches had low associativity. In these earlier works, packing is referred to as data copying because no data-layout change is needed to obtain this effect [79]. Second, and currently more relevant as problem sizes grow, fewer TLB entries are required for the address translations of a packed subtensor. Third, better vectorization. By employing data-layout changes in the packing copy, packing can enable the compiler to use vector instructions to access the packed subtensor, as its elements are rearranged to the order in which they are accessed. Vectorization would be much less efficient if the elements were copied into the packed subtensor in the same order as they were stored in the original tensor.

As the rearranged elements follow access order in memory, stream prefetching may also become more effective.

For example, assume that A is a row-major matrix and A' is a submatrix of A as in (2.1). If the elements a, e, b, and f of A' are accessed in that order, then fewer TLB entries may be required to address these elements if they are copied into a separate memory region in the same order as in A. Still, elements of A' will not be contiguous with respect to the access order a, e, b, and f. These elements can be made contiguous by packing them as in A'^T in (2.2). Copying elements of A' into A'^T requires accessing A in an order with poor cache locality and thus higher latency. Nevertheless, the added latency during the packing copy is compensated by the lower latency when reusing the elements in A'^T. The sequential access to submatrices such as A' is common in computations involving matrices after being transformed by loop tiling.

$$A = \begin{bmatrix} \overbrace{\begin{bmatrix} a & b \\ e & f \end{bmatrix}}^{A'} & c & d \\ g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \qquad (2.1) \qquad A'^T = \begin{bmatrix} a & e \\ b & f \end{bmatrix} \qquad (2.2)$$

Goto *et al.* [33] describe high-performance CPU implementations of GEMM, where packing is a crucial step. Packing is applied to reduce the number of TLB entries required by submatrices such that the TLB does not become a limiting factor for the computation. In their work, packing also changes the data layout of the packed submatrices so that consecutive operations access consecutive data in memory. As a result, data is more easily loaded to registers due to the increased spatial locality. Many state-of-the-art linear algebra libraries such as Eigen [36], OpenBLAS [88], and BLIS [81] follow the strategy described by Goto *et al.* in their implementations of GEMM. Aside from a few works where packing can be applied as a user directive or where it is part of an auto-tuning strategy [5], [84], [87], these library implementations of GEMM are the extent of current packing use.

Though packing offers the benefits previously mentioned, deciding which tensors to pack and at which point in a loop nest to place the packing of a tensor is difficult. Moreover, if the computation modifies the packed subtensor, then it needs to be *unpacked* — copied back to the original tensor. Unpacking creates an additional copying overhead. The copy-operation overhead can outweigh the benefits of packing.

Thus, packing should only be applied when its performance gains are greater than such overhead. As discussed by Lam *et al.* [49] and Temam *et al.* [79], the copying overhead of an unrestricted packing algorithm that packs all tensors can easily outweigh its gains.

2.2 Packing Analysis

The main contribution of this chapter is GPAT, a packing analysis and code transformation that generalizes the idea of packing. In contrast to existing packing approaches, GPAT is more robust to changes in tiling strategy and loop order. In fact, to GPAT, loop tiling is an optional prior transformation: any computation with loop-nest structures of at least depth three may contain valid packing opportunities. Nevertheless, prior loop-tiling transformations may introduce additional packing opportunities for GPAT to identify.

2.2.1 Preliminary Definitions and Notation

The presentation of GPAT assumes an Intermediate Representation (IR) where a `for` loop is represented as an operation encoding the Induction Variable (IV), upper and lower bounds, increment step, and body of the loop. The body of a `for`-loop operation may contain other operations, including other `for` loops. In this IR, a `for` loop is a Single-Entry Single-Exit (SESE) region in a CFG.

In this IR, a contiguous region of memory can be represented as a n-dimensional *tensor*. A tensor element is accessed by indexing each of its dimensions separately with the elements of an *index tuple*. The address of a tensor element is given by the sum of the tensor base address and a linear combination of the index and stride for each dimension. Each element of an index tuple can be an affine expression of loop IVs, constants, and variables in the program. The *shape* of a tensor is defined by the length of each of its dimensions.

The *footprint* of a tensor is the number of bytes the tensor occupies in memory; it is calculated by the product of the length of each dimension and the size in bytes of the element data type of the tensor. The *working set* of a tensor T in a `for`-loop L is a subtensor formed by the set of elements of T that are accessed inside L.

Packing a *tensor* T in a *target loop* L involves three steps: (i) Insert a packing loop immediately before the entry block of the SESE region of L. This packing loop creates T', a tensor containing a copy of the working set of T in L, and may change the data layout of the working set of T in T'. (ii) Substitute all memory references to T by references to T' in L. (iii) If there are writes to T', then insert an unpacking loop immediately after the exit block of L. Unpacking copies the elements of T' to their respective positions in T. A loop-tensor pair (L, T) represents a *packing candidate*.

The packing loop may swap elements of the indexing tuple of T' to change the order in which elements are copied into T'. This data-layout change is represented with a *permutation vector* of size n, where n is the dimensionality of T. The identity permutation vector $[0, 1, 2, \ldots, n-1]$ represents no data layout change. Each element i corresponds to the i-th dimension of tensor T. Swapping elements of the identity creates a data-layout-altering permutation.

2.2.2 Analysis Constraints

Consistent with the practice in widely adopted IRs, GPAT's analysis requires `for` loops to be Static Control Parts (SCoPs) [48] of a program and the shape of tensors to be statically known. The idea is that an earlier pass can convert computations with unknown tensor shapes into fixed-sized tensor tiles — a practice made more relevant with the proliferation of hardware accelerators that compute on fixed-sized operators. Given these requirements, the shape of a tensor's working set in a loop is also statically known.

2.2.3 Running Example

The 3-dimensional tensor contraction represented in a pseudo-IR code in Listing 2.1 exemplifies GPAT in this section. In this non-tiled example, the input tensors are $\mathtt{A}_{80\times100\times50}$, $\mathtt{B}_{100\times80\times60}$, and $\mathtt{C}_{50\times60}$, where `C` was initialized to zero.

Listing 2.2 shows a packing transformation applied to Listing 2.1. Loops are referred to by the name of their IV, *e.g.* $\mathtt{for}_J$ is the loop in Line 3 of Listing 2.1. Listing 2.2 highlights the packing loop applied to tensor `A` targeting $\mathtt{for}_J$ and the load to the packed subtensor `A'`. This packing applies a data-layout change by reordering

```
1                       // A_{80x100x50}
2   for(i=0; i<50; i++) { // A_{80x100x1}
3     for(j=0; j<60; j++) { // A_{80x100x1}
4       for(k=0; k<80; k++) { // A_{1x100x1}
5         for(l=0; l<100; l++) { // A_{1x1x1}
6           a = load A[k][l][i]
7           b = load B[l][k][j]
8           prod = mul a, b
9           c = load C[i][j]
10          sum = add c, prod
11          store sum, C[i][j]
12  }}}}
```

Listing 2.1: 3D Tensor Contraction: $C_{i,j} = \sum_k \sum_l A_{k,l,i} \times B_{l,k,j}$. The shape of the working set of A is shown in comments outside of the loop nest and inside each loop.

the index tuple of `A'` in Line 6 from `[m][n][0]` to `[0][m][n]` corresponding to the permutation vector `[2,0,1]` (from the identity `[0,1,2]`). The change is trivial because the length of the outermost dimension of `A'` is 1. As a result, `A'` has a shape of $(1, 80, 100)$. The working-set shape of `A` is shown for every `for` loop in Listing 2.1, similarly, the working-set shape of `A'` is shown in Listing 2.2.

2.2.4 Analysis Overview

GPAT's analysis receives a loop nest as input. In a given loop nest, multiplying the number of contained `for` loops by the number of tensors accessed gives the total number of possible packing candidates. A tuning strategy would have to consider all subsets of the set of packing candidates, excluding redundant ones. A packing candidate subset is redundant if two candidates pack the same tensor in different target loops, but one loop contains the other. In contrast to tuning, GPAT's analysis statically determines its output: a profitable selection of packing candidates and their respective data-layout changes if any. To do so, GPAT uses the following architecture-specific parameters: (i) Size of the L1, L2, and L3 caches (KiB); (ii) Number of L1 data TLB entries and (iii) Size of a page addressed by these entries (KiB).

GPAT's analysis has four phases. The first and second *filtering* phases narrow the pool of packing candidates, forwarding only candidates that exhibit data reuse and maintain cache residency to the subsequent phases. The third phase, *goal fulfillment*, forwards remaining candidates that fulfill at least one of two packing

```
1  for(i=0; i<50; i++) {
2    A' = alloc(1x80x100) // A'_{1x80x100}
3    for(m=0; m<80; m++) {
4      for(n=0; n<100; n++) {
5        tmp = load A[m][n][i]
6        store tmp, A'[0][m][n]
7    }}
8    for(j=0; j<60; j++) {  // A'_{1x80x100}
9      for(k=0; k<80; k++) { // A'_{1x1x100}
10       for(l=0; l<100; l++) { // A'_{1x1x1}
11         a = load A'[0][k][l]
12         b = load B[l][k][j]
13         prod = mul a, b
14         c = load C[i][j]
15         sum = add c, prod
16         store sum, C[i][j]
17 }}}}
```

Listing 2.2: Packing applied to 3D Tensor Contraction. The working set of `A'` is shown in comments.

goals: (i) innermost access stride reduction and (ii) TLB miss reduction. These goals prevent GPAT from selecting candidates for which packing benefit is outweighed by packing overhead. Though all candidates in the resulting pool fulfill the analysis goals, certain candidate subsets may contain redundant candidates. Thus, the final *selection* phase defines a cost-benefit function and greedily selects a non-redundant candidate subset from Phase 3 candidates that maximize cost-benefit.

Phase 1. Data Reuse Filter. This phase eliminates a candidate (L, T) if the packing-loop-created subtensor T' is not reused across iterations of L; a requirement to overcome packing-loop overhead. If an access is invariant to the IV of loop l, the same set of elements are accessed on each iteration of l and are reused.

This filter checks if all memory instructions related to tensor T in loop L are invariant to the IV of L. A flow analysis is required to determine loop invariance because an IV i_2 may depend on another IV i_1 if the loop that defines i_2 uses i_1 in its lower or upper bound expressions. This dependency relation is transitive: an IV i_3 that depends on i_2 also depends on i_1. Such dependencies must be checked for the IVs of each child loop of L — a common case on tiled loops. Thus, a subtensor T' of a tensor T is reused in the iterations of a target loop L if and only if the index tuple

of each instruction accessing T' depends neither on L's IV nor any IV depending on L's IV.

Listing 2.1 has four candidates (L, T) that exhibit reuse: {($\texttt{for}_J$, A), ($\texttt{for}_I$, B), ($\texttt{for}_K$, C), ($\texttt{for}_L$, C)}. In the ($\texttt{for}_J$, A) candidate, which is packed in Listing 2.2, all elements of A' are reused in every iteration of $\texttt{for}_J$ because the load to A in Line 6 of Listing 2.1 is invariant to the IV j.

This phase is conservative and has a simple notion of reuse; certain cases where it may be beneficial to pack could be filtered out. However, the modularity of the design allows for future integration of a more sophisticated reuse analysis.

Phase 2. Cache Residency Filter. The second step of GPAT's analysis is to filter packing candidates (L, T) based on cache residency. If a packed tensor T' is reused in L, but there is not enough space available in the cache, T' could be evicted between iterations of L, resulting in cache misses and increased access latency to T'. To avoid this, the second filter eliminates candidates whose packed tensor cannot remain cache resident while in use.

Modern CPUs have multiple levels of cache. A packed tensor T' may remain resident in one level while evicted in another. GPAT sets a target cache level for its input loop nest where packed tensors should remain resident. The target cache level is selected as the largest cache that cannot store the total footprint of all tensors accessed in the loop nest. For example, the total footprint of the tensor contraction in Listing 2.1 is the sum of the footprints of A, B, and C. Lastly, if the total footprint fits in L1, then all data can remain in L1 during the computation. In this case, TLB misses are not likely limiting the performance and packing is not applied.

A candidate's packed tensor T' can remain cache resident if the target cache is large enough to hold: (i) the footprint of T' and (ii) twice the footprint of the working set of all other tensors in one iteration of the target loop L. The reuse distance between two accesses to the same element of T' is the number of accesses in one iteration of L. After completing the number of accesses in this distance, elements of T' are brought back to cache. This distance indicates that having space in the cache for T' and the working set of all other tensors in one iteration of L would ensure the residency of T'. However, unlike the working set of T', the working set of the other

tensors may change between iterations of L. Accesses to T' can also be interleaved with accesses to other tensors. As a result, in a Least Recently Used (LRU) cache policy, elements of T' could be the least recently used. Thus, twice the footprint of the working set of all other tensors ensures that T' can remain resident because this is enough space not only for the i-th iteration, but also for the $(i+1)$-th iteration. GPAT generalizes Mitchell *et al.*'s idea on ensuring residency for tiled GEMM. [66].

Loop L may have multiple immediate child loops and conditional statements. Tensor accesses may belong to mutually exclusive blocks of code selected by a conditional statement. For those cases, the GPAT's analysis may over-approximate the required cache size to ensure residency. However, even being conservative in this phase, GPAT shows substantial speedup for *2mm* benchmark tiled with Polymer, which has mutually exclusive if conditions (see Section 2.3.3).

Assume Listing 2.2 targets the L2 cache. Elements of the packed tensor `A'` have a reuse distance equivalent to the number of accesses in one iteration of the target loop $\texttt{for}_J$. In one iteration, all elements of `A'`, and a working set of shape $\texttt{B}_{100\times80\times1}$ and $\texttt{C}_{1\times1}$ are accessed. To ensure L2 residency of `A'` throughout the iterations of $\texttt{for}_J$, the L2 should have space for `A'` and twice the mentioned working-set footprint of `B` and `C`. Such is the footprint of two $\texttt{for}_J$ iterations.

Phase 3. Goal Fulfillment. After the filter phases, GPAT's analysis identifies which Phase-2-passing candidates fulfill at least one of the two goals detailed in the following paragraphs. Goal fulfillment indicates that selecting a candidate will benefit performance.

Prior to verifying goal fulfillment, GPAT's analysis checks each candidate for opportune data-layout changes to minimize the stride of consecutive accesses to the packed tensor.

The stride-minimizing data-layout change of an access is obtained by: (i) listing the depth of loops that create the IVs used in the index tuple of the access; (ii) saving the maximum depth for each element of the index tuple and (iii) permuting the elements of such index tuple so that elements with lower loop nest depths precede elements with higher depths. This permutation can be applied to the packing loop of T', and consequently to all accesses to T', to change its data layout. The

permutation vector representing this data layout change is saved for the candidate. If no permutation is needed, or if the permutation order that minimizes stride is not the same for all accesses to T', the permutation vector is the identity. The permutation of A in Listing 2.2 is discussed in Section 2.2.3.

Innermost Access Stride Reduction. Reducing the stride between iterations of innermost loops improves cache locality and can enable vectorization. Thus, this goal is fulfilled by a candidate if a data-layout change of T' reduces stride between accesses to T' in two consecutive iterations of an **innermost loop**. Stride reduction between iterations of non-innermost loops mainly reduces the number of TLB entries required to access T' and is considered in the next goal. Additionally, to fulfill this goal a candidate requires the innermost loops to access the equivalent of at least two cache lines of packed tensor elements. This criterion is in place to avoid retaining candidates whose performance benefit is outweighed by packing overhead.

In the running example, the packing candidate applied to Listing 2.2 changes data layout of `A'` with a permutation vector $[2, 0, 1]$. Contrasting with the 50-strided accesses to `A` across iterations of the innermost loop $\texttt{for}_L$ in Listing 2.1, the data-layout change of `A'` ensures that accesses have unit stride across consecutive iterations of $\texttt{for}_L$ in Listing 2.2.

TLB Miss Reduction. A loop may suffer from TLB capacity misses if the number of entries needed to access data within the loop exceeds TLB capacity. A packed subtensor, with a possible data-layout change, requires less TLB entries to resolve address translations. This goal is fulfilled if a candidate's packed subtensor reduces the number of required L1 data TLB (dTLB) entries below or to L1 dTLB capacity.

This phase uses three functions: (i) `wSS(t, l)` returns the working-set shape of tensor `t` in loop `l`; (ii) `perm(PV, s)` applies the permutation vector `PV` to a shape `s`; (iii) `estTLB(`s_{ws}`, s)` estimates the number of TLB entries needed to address a working set of shape s_{ws} in a tensor of shape s. Using these functions, Listing 2.3 describes a Boolean function to determine goal fulfillment.

For a candidate (L, T) with permutation vector PV in an L1 dTLB with `TLBEntries` entries, for L and each of its child loops, Listing 2.3 estimates the

```
1  def improvesTLB(L, T, PV, TLBEntries):
2    packedTShape = perm(PV, wSS(T,L))
3    for l in {L, child loops of L}:
4      Packing = 0, NoPacking = 0
5      for t in {tensors accessed in l}:
6        Entries = estTLB(wSS(t,l), shape(t))
7        NoPacking += Entries
8        if t == T:
9          Packing += estTLB(perm(PV,wSS(t,l)), packedTShape)
10       else:
11         Packing += Entries
12     if NoPacking > TLBEntries and Packing <= TLBEntries:
13       return true
14   return false
```

Listing 2.3: Function that checks if a packing candidate achieves the goal of reducing TLB misses.

number of TLB entries needed to translate the working set of tensors accessed — for both packing and not packing the candidate. The variables `NoPacking` and `Packing` store the TLB entry estimates for each of these loops. Only the number of TLB entries required to access T is affected by packing (L, T). For all other tensors, this number is computed in Line 6 by estimating the entries needed to access a working set of the shape of `t` in `l`, given `t`'s shape is `shape(t)`.

Line 9 approximates the number of TLB entries required to access T in a loop if the working set of T in L is packed into T'. In this line, both shape parameters to `estTLB` are permuted by `PV` of the candidate. The second parameter is the shape of T' instead of the shape of T.

For example, in Listing 2.1, `A` has a working set of shape $\texttt{A}_{1\times100\times1}$ in $\texttt{for}_K$, whereas, in Listing 2.2, `A'` has a working set of shape $\texttt{A'}_{1\times1\times100}$ in the same loop. Recall that the shape of `A` is $\texttt{A}_{80\times100\times50}$, and that the candidate packed in Listing 2.2 is $(\texttt{A}, \texttt{for}_J)$. Assume an L1 dTLB that contains 64 entries, and for simplicity, assume that each entry can address 50 elements of `A`. In this setting, 100 TLB entries are needed to access the working set of A in $\texttt{for}_K$, whereas by packing `A` into `A'` the same working set requires only two entries. This example would be run in the `improvesTLB` function of Listing 2.3 when `L` is $\texttt{for}_K$ and T is `A`. The TLB entries of the other tensors would also need to be estimated to check the `if` condition in Line 12.

Phase 4. Greedy Selection To determine which packing candidates to select, GPAT's final phase performs a cost-benefit analysis for each candidate that fulfills at least one of the goals in Phase 3. The *benefit* of a candidate is quantified by the reduction in the number of TLB entries from packing T' and from the reuse of T'. The benefit is the linear combination of the TLB entry reduction for every loop in which Line 12 in Listing 2.3 evaluates to true and the number of times the corresponding loop is executed. Moreover, the *cost* estimates packing overhead with the footprint of T' — and if T' is stored to, twice the footprint of T. GPAT treats the ratio of benefit to cost of each candidate as a proxy for performance improvement, providing a means of sorting candidates from most to least beneficial. If two candidates have an equivalent cost-benefit ratio, GPAT resolves ordering by comparing the depths of the target loops; a smaller depth persists T' for more of the computation and suggests higher reuse of elements in T'. Once sorted, GPAT iterates through the list, greedily selecting candidates. To be selected, a candidate: (i) cannot be redundant with respect to the set of committed candidates; (ii) must continue to fulfill either of the goals in Phase 3 when considered with the set of committed candidates. To verify (ii), before selecting a candidate, the `improvesTLB` function is rerun with the knowledge of the committed candidates.

2.2.5 GPAT in the MLIR Affine Dialect

MLIR [52] is a recent addition to the LLVM compiler infrastructure project [50]. Its goals are to extend the levels of representation possible in LLVM and to be a common framework for high-level abstractions. MLIR is a combination of IR dialects, as opposed to LLVM's general-purpose IR. Affine is one of such dialects, providing an IR that is amenable to polyhedral analysis by preserving high-level loop-nest structures that are available in languages such as C.

Most high-level concepts used in the GPAT's analysis description are present in the Affine dialect. Tensors are represented by `memrefs`, which are accessed through index maps. `for`-loops are first-class operations. As GPAT's implementation builds upon Affine's *data copy generation* transformation pass, utility functions that compute tensor shape and footprint, and the working set shape of a tensor in a loop are already defined. Like Affine's loop tiling, GPAT's Affine implementation is available

as a modular pass within the infrastructure and will be available as an artifact.

2.3 GPAT Evaluation

This section evaluates GPAT using its MLIR Affine implementation. As a modular compiler analysis and transformation, GPAT applies packing to varied input loop nests, showing the generality of the approach beyond GEMM. The evaluation shows that GPAT can be applied to the output of the state-of-the-art Pluto [6] loop optimizer and obtain considerable speedup. Moreover, even when applied to non-optimized loop nests, or only tiled nests, GPAT can surpass the performance of the Polly [35] loop optimizer.

The evaluation addresses the following questions:

① Can GPAT select an effective combination of packing candidates?

② How does the performance of GPAT compare to previous loop optimization approaches?

③ Is GPAT robust to prior loop transformation strategies?

④ Is GPAT general? For instance, can it discover packing opportunities in computations other than GEMM?

⑤ Can GPAT reduce TLB pressure and increase opportunities for vectorization?

2.3.1 Setup

The machine used in the evaluation of GPAT runs Ubuntu 20.04.1 (Kernel 5.15.0-46) and is equipped with an Intel i5 8500 locked at 3.0 GHz and 32 GiB of DDR4 memory. The evaluation benchmarks are written in the C language; namely, programs from the PolyBench/C[1] benchmark suite [73]. PolyBench/C has thirty numerical computations including linear algebra kernels and solvers, data mining algorithms, and stencil computations. Its main objective is to provide a set of representative computations to evaluate polyhedral-optimization approaches. Many of the programs have deep loop nests that provide a suitable input for evaluating GPAT. All the experiments use the large dataset size of Polybench/C. At the time of writing, there is no mature tool that compiles C/C++ programs to MLIR. This evaluation employs

[1]Version 4.2.1.

Polygeist[2], a research C/C++ frontend that generates MLIR SCF & Affine Dialect code [67] to compile C programs to MLIR.

Due to MLIR's current lack of parallel code generation mechanisms, each experiment was single-threaded. Multi-threaded code could affect GPAT's decisions, but it is not explored at this time.

2.3.2 Experimental Methodology

The results presented in Section 2.3.3 used the Google Benchmark[3] tool [32] to collect the average execution time of 100 executions of each evaluated approach. Section 2.3.3 employs the above methodology to evaluate:

(i) **Polymer**[4]: an MLIR tool that extracts SCoPs of a program and optimizes the loops within each SCoP using Pluto [67], a polyhedral optimizer. The goals of Pluto are to improve locality and parallelism through loop-nest transformations including loop interchange, loop fusion, loop skewing, loop reversal, and loop tiling;

(ii) **GPAT**[5]: packing applied by GPAT, which takes Polymer-optimized IR as input; and

(iii) **Individual Packings**: packing applied by GPAT individually to all candidates that exhibit reuse; identified by Phase 1 of GPAT's analysis in the IR optimized by Polymer.

These approaches were evaluated across a range of tiling factors $t = 8, 10, 12, \ldots, 512$ that were supplied to Polymer. All loops shared the same tiling factor.

Section 2.3.4 employs a similar methodology but uses the built-in Polybench/C timing facilities. The following approaches are evaluated in Section 2.3.4:

(i) **Polly**[6]: an LLVM IR polyhedral loop optimizer [35];

(ii) **Clang -O3**: Clang[6] compilation with its optimization level 3 (-O3);

(iii) **Polymer**;

(iv) **Affine**: the MLIR Affine dialect tiling pass and

[2]Commit 6ba6b7b8ac07c9d60994eb46b46682a9f76ea34e.
[3]Version 1.6.1.
[4]Commit 4bb0aa2ff43f70bb3f13f221ebcf9f76fb41fa76.
[5]Commit packing-v0.5
[6]Version 14.0.1

(v) **GPAT**: packing applied by GPAT, with either Polymer-optimized or Affine-tiled IR as input.

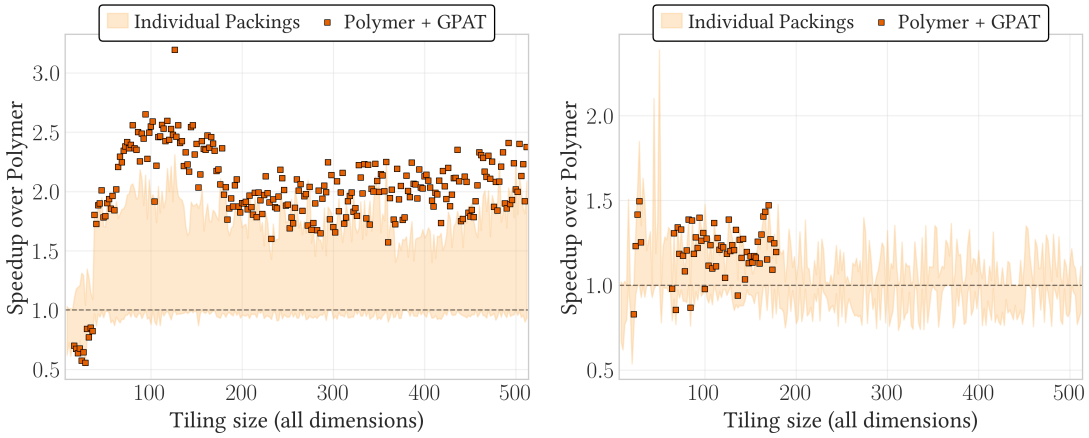
With **Affine**, rather than running benchmarks at each tiling factor, the tiling pass selects a factor based on a target cache-size parameter; in this case, the sizes for the L1, L2, and L3 caches. Polly does not take an input tiling factor, so it is run with a single configuration.

Because GPAT is orthogonal to tiling, the size and shape of tiles are decided prior to GPAT. Therefore, the graphs only show a result for GPAT in the tile sizes that it determined that packing should be applied.

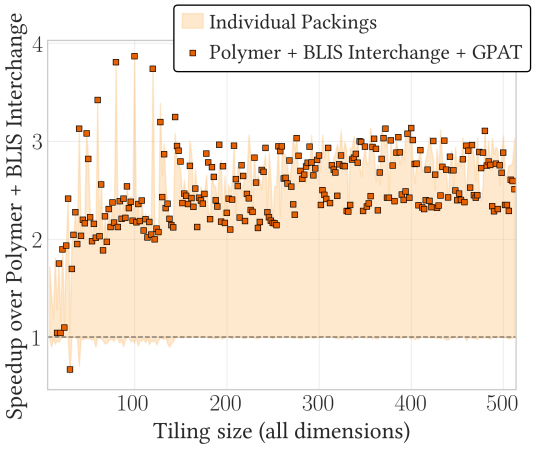
For all evaluated approaches the floating-point contract used is `fast-math`, pointers are declared as non-aliases, and the compilation flags used are `-march=native -flto`. Without declaring pointers as non-aliases, GPAT would have a slight advantage as a compiler can easily determine that a packed tensor is not an alias. For **Polly**, pattern-matching-based optimizations are disabled and cache associativity and size information are passed as parameters. **GPAT** uses the architecture-specific parameters described in Section 2.2.4.

2.3.3 Packing Selection Evaluation

This experiment shows the execution time of **GPAT** and **Individual Packings** relative to **Polymer** to address ①. For each tile size, the performance range from the lowest to the highest performing **Individual Packings** is shown as the light orange area in the graphs. That is, vertical slice of the light orange range is composed of multiple points that are individually measured, the range is shown instead of the points for clarity. The higher end of the ranges in Figures 2.1a and 2.1c indicate that most packing candidates in these benchmarks are beneficial for performance. The performance of **GPAT** is shown by the orange squares in the graphs for the tile sizes for which GPAT's analysis deemed packing to be beneficial. **GPAT** may select a combination of packing candidates, thus falling outside of the range of **Individual Packings**. The performance obtained by combining individual packing candidates answers ① affirmatively for most, but not all, tile sizes.



(a) *2mm* (b) *gemm*



(c) *gemm* with BLIS Loop Ordering

Figure 2.1: Packing selection evaluation graphs

2mm.

Figure 2.1a reports on the *2mm* benchmark which computes two matrix multiplications and uses a total of five matrices. The second multiplication has an operand that is the result of the first matrix multiplication. *2mm* can be expressed as $D_{M\times N} = \beta * D_{M\times N} + T_{M\times N} * C_{N\times N}$, where $T_{M\times N} = \alpha * A_{M\times K} * B_{K\times N}$. Although matrix multiplication is the core of this benchmark, *2mm* behaves differently from GEMM because there is a data dependence between the two matrix multiplications. After *2mm* is optimized by **Polymer**, the two level-3 loop nests that represent the two matrix multiplications are tiled, interchanged, and partially fused into two loop nests. The first loop nest is a level-4 nest that initializes $T_{M\times N}$ with zeros and computes $D_{M\times L} = \beta * D_{M\times L}$. The second loop nest is a tiled level-6 nest that fuses

the computation of $T_{M\times N} = \alpha * A_{M\times K} * B_{K\times N}$ and $D_{M\times L} \mathrel{+}= T_{M\times N} * C_{N\times N}$. This second loop has three mutually exclusive if conditions in its level-3 loop. GPAT does not find any profitable opportunities in the first loop nest, but in the second, it identifies 13 individual packing candidates that pass Phase 1. Polymer's optimization trades a unit-strided access order between iterations of its innermost loops for matrix `B` to fuse the two multiplications. The fused loop has multiple opportunities for packing that change the data-layout of B, making its accesses unit strided in the innermost loops. Additionally, many TLB entries are needed to address the tiles of the five matrices involved in the fused loop. Thus, packing some of these tiles can also be beneficial to avoid TLB misses. Overall, GPAT packed from 0 to 4 of the 13 individual packing depending on the tile size. Figure 2.1a, indicates an affirmative answer to ①: for the majority of tile sizes, GPAT selects an effective set of packing candidates, surpassing the performance of **Individual Packings** and **Polymer**.

The slowdown observed with tile sizes between 8 and 36 is attributed to the LLVM loop unrolling pass. When disabling loop unrolling, the observed speedup follows the trend of the higher tiling sizes. Of the thirteen candidates that compose **Individual Packings**, GPAT's code transformation that relies on MLIR Affine's infrastructure fails for two candidates because it computes incorrect loop bounds, these candidates were omitted from the graph.

gemm.

Figure 2.1b reports on the *gemm* benchmark, calculating a single matrix multiplication expressed as $C_{M\times N} = \alpha * A_{M\times K} * B_{K\times N} + \beta * C_{M\times N}$. Polymer tiles and interchanges loops, producing well optimized code that already ensures a unit-strided access order to the innermost dimension of its tiles. Thus, no data-layout changes need to be applied to the packing candidates of this benchmark. For this benchmark, GPAT selects packing candidates based only on reducing TLB misses and Figure 2.1b shows orange squares only for points where GPAT determines that a beneficial TLB-entry reduction can be achieved. For most tiling sizes for which GPAT applied packing, a tile of matrix `B` is packed. Given the loop ordering used by **Polymer**, the performance range of **Individual Packings** shows that for some tile sizes, packing only deteriorates performance in *gemm*. Even so, **GPAT** surpasses the

performance of **Polymer** on the majority of tile sizes for which it applied packing. In Figure 2.1b, the orange squares of **GPAT** are all within the range of **Individual Packings** because GPAT selected only one candidate.

***gemm* with BLIS Loop Ordering.**

Figure 2.1c reports on the GEMM benchmark using the loop nest ordering suggested by the BLIS framework [81] — different from the loop order generated by Polymer and evaluated in the previous section. Such loop order is based on the work by Goto *et al.* [33], and it is used in BLIS in combination with packing to optimize *gemm*. **GPAT**'s performance is shown w.r.t. BLIS loop ordering and without packing in Figure 2.1c. GPAT found data-layout transformation opportunities that result in simpler memory access patterns — by reducing strides of consecutive accesses — in the innermost loop. In addition to reducing the number of TLB entries to address a tile, **GPAT** obtained consistent performance improvements across tiling sizes. Examining this alternative loop nesting order revealed better opportunities for packing. The contrasting results of Figure 2.1b and Figure 2.1c highlight the importance of building a modular packing compiler pass that is agnostic to decisions made by other compiler passes, and thus indicate an affirmative answer to ③.

TLB Misses and Vectorization.

To address ⑤, hardware performance counters were collected on the *2mm* benchmark from the experiment outlined in Section 2.3.3. For each tile size in the experiment, the analysis used the Linux `perf` tool to collect counters for: (i) L1 dTLB load misses, more precisely, L1 dTLB load misses that were a hit in the L2 TLB, shown in Figure 2.2; and (ii) total instruction count, shown in Figure 2.3. As opposed to the previous graphs, these figures show the average counter values for the 100 iterations, therefore lower values represent fewer L1 dTLB misses or fewer instructions executed. For a given tile size, **Individual Packings** is represented by the highest and lowest value of the hardware performance counter obtained by an individual candidate, shown as the light orange area in the graphs.

Through Phase 3's TLB-aware analysis, GPAT can find and select packing candidates that lower TLB requirements of a loop and reduce TLB misses substantially.

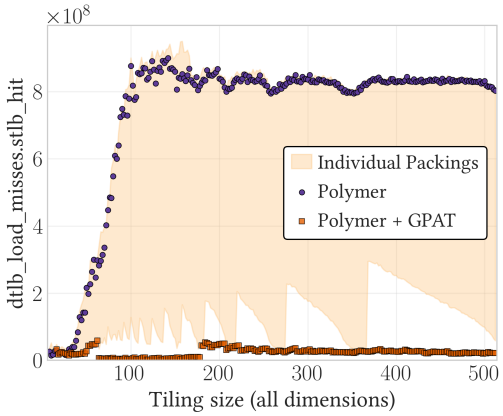


Figure 2.2: *2mm* L1 dTLB load misses.

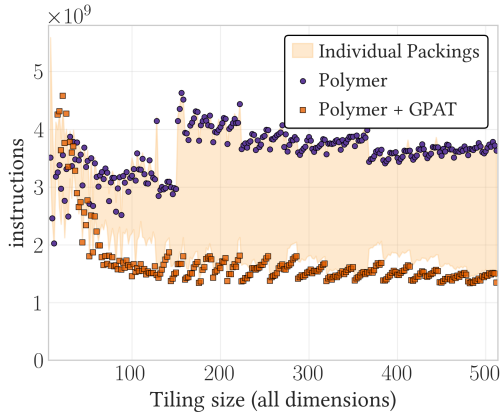


Figure 2.3: *2mm* instructions count.

Figure 2.2 shows that the reduction in TLB misses for the candidates selected by GPAT is more significant than for **Individual Packings**.

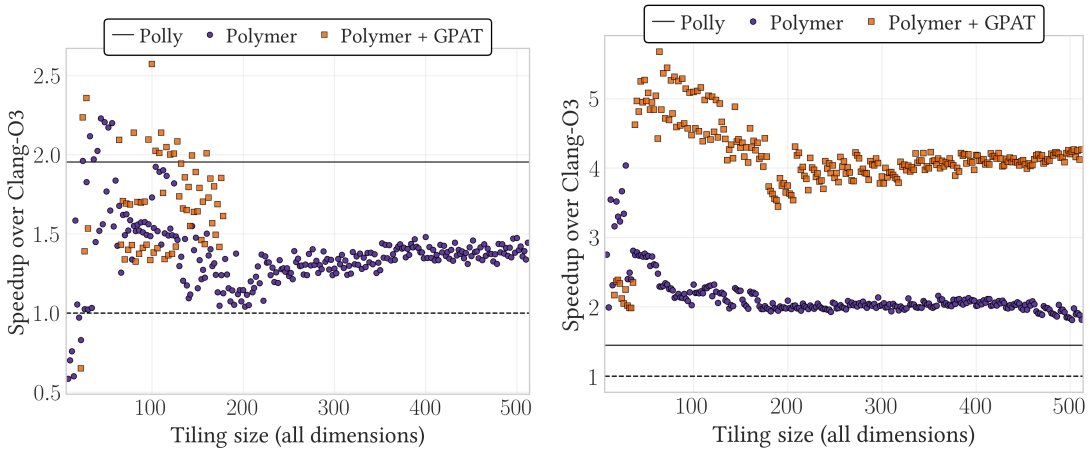
The packing and unpacking of buffers increases the number of instructions, but by transforming non-contiguous accesses into contiguous accesses — through data-layout changes within the packed buffer— **GPAT** is able to better utilize CPU vector instructions, leading to a decrease in total instruction count. Specifically, in *2mm*, increased vectorization is afforded by the data-layout change of two packing candidates that pack a tile of matrix `B` in the second loop nest of the computation described by Section 2.3.3. GPAT selects these candidates in most tile sizes, leading to the instruction count reduction shown in Figure 2.3. The leftmost tiling sizes are the exception, similarly to the speedup graphs, these tile sizes are affect by loop unrolling, leading to an increase in instruction counts.

Together, Figures 2.2 and 2.3 indicate an affirmative answer to ⑤: GPAT can reduce TLB misses and improve vectorization.

2.3.4 PolyBench Evaluation

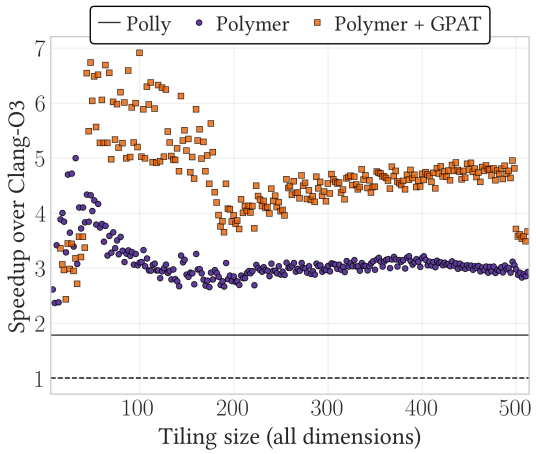
This experiment addresses questions ②, ③, and ④ by comparing **GPAT** against other loop optimization approaches and evaluating it beyond GEMM. The experiment used two tiling engines, **Polymer** and the **Affine**. Unlike Polymer which can also apply optimizations such as loop interchange and loop fusion, the Affine tiling pass can only perform tiling and is restricted to perfectly nested loops.

This experiment contrasts the results of **Polly** with the results of **GPAT** applied



(a) *gemm*, baseline time: 446 ms

(b) *2mm*, baseline time: 1673 ms



(c) *3mm*, baseline time: 3140 ms

Figure 2.4: Polybench evaluation on the Polymer tiling engine

after each of the two tiling engines. All results are shown as speedup over the baseline **Clang -O3**. The experiment considers each benchmark in the PolyBench/C suite and the results are reported for each of the benchmarks in which GPAT discovers and selects opportunities for packing. Polybench was engineered as an optimization target for Polly. Therefore, results that show **GPAT** outperforming **Polly** evidence GPAT's effectiveness.

Polymer Tiling Engine.

As shown in Figure 2.4, GPAT identifies optimization opportunities in *gemm*, *2mm* and *3mm*. For *2mm*, **GPAT** identifies opportunities for data-layout changes after a partial loop fusion performed by Polymer (Section 2.3.3). The code generated

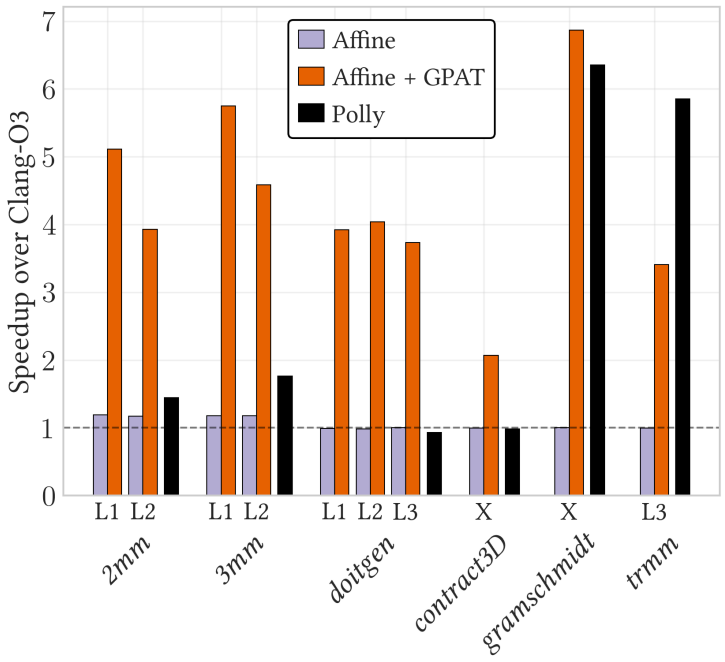


Figure 2.5: Polybench evaluation on the Affine tiling engine. Baseline times (ms) of each benchmark in the same order of figure: 1673, 3121, 482, 93, 13919, and 2966

with GPAT is consistently faster than the code generated by **Polly**, which optimizes for data access pattern but does not apply data-layout transformations: in *2mm* there is no way to reduce the stride of all tensor accesses only by applying loop interchange. Polly is optimized well for *gemm*. Still, **GPAT** outperforms **Polly** at multiple tiling sizes by reducing the number of TLB entries required to access *gemm*'s inputs (Section 2.3.3).

The *3mm* benchmark computes a matrix multiplication ($G_{M\times L} = E_{M\times N} * F_{N\times L}$), where each operand matrix is itself the result of a matrix multiplication ($E_{M\times N} = A_{M\times K} * B_{K\times N}$ and $F_{N\times L} = C_{N\times P} * D_{P\times L}$). Syntactically, this benchmark contains the computations for both *gemm* from Section 2.3.3 and *2mm* from Section 2.3.3 after optimized by Polymer. GPAT selects and combines the packing candidates that are beneficial to both *gemm* and *2mm* into one computation, achieving the highest speedup overall. Polly fails to reduce the memory-access stride because it does not apply data-layout changes. The slower performance for small tile sizes of *2mm* and *3mm* are attributed to LLVM's loop unrolling.

Affine Tiling Engine.

Figure 2.5 shows the results from applying GPAT to the output of Affine. The label under each pair of columns indicates the cache level used for the cache-size parameter in the **Affine** tiling transformation. An X indicates that tiling did not change the output. Results are reported only for cache-level sizes for which GPAT selected packing candidates. GPAT identifies packing opportunities in five benchmarks from the PolyBench suite and in *contract3D*, the running example described in Section 2.2.3.

For *gramschmidt*, a solver that computes the QR Decomposition by applying the Gram-Schmidt process to the set of column vectors in a full-rank input matrix, results exemplify the generality of GPAT: *gramschmidt* does not resemble a GEMM, thus confirming the decoupling of packing from handcrafted GEMM implementations and affirmatively answering ④. Moreover, Affine fails to tile *contract3D* and *gramschmidt*, providing evidence that GPAT is independent of tiling transformations.

GPAT also discovers opportunities in *trmm* and *doitgen*, which are not packed if their loops are optimized by Polymer. These latter observations provide affirmative support for ③: GPAT optimizes computations with loop nests if potential benefit is found, regardless of the computation and prior transformations. The performance of *trmm* produced by Affine tiling is the same as **Clang-O3** because it tiles only one loop. Tiling a single loop — equivalent to strip-mining — does not affect the overall execution of a loop nest. GPAT improves on the Affine-produced *trmm* and outperforms Polly on *contract3D* and *gramschimidt* in a similar scenario. However, solely applying packing to the otherwise unoptimized *trmm* is not sufficient to outperform the loop optimizations — including tiling strategy — that Polly applies.

2.4 Discussion

This chapter presents GPAT, a modular compiler analysis and code transformation to determine what and where to apply packing. Using an analytical model that accounts for TLB utilization and data-layout changes, GPAT decides when applying packing improves performance. Although the abstractions in MLIR are convenient to implement GPAT, GPAT can be integrated into other production-ready compilers

(*e.g.* LLVM) to automatically optimize computations beyond GEMM while being orthogonal to tiling strategy. Even though GPAT by itself surpasses the performance of current loop optimizers for some benchmarks, it can also be used alongside them to improve overall performance.

This research began with the observation that modern loop optimizers often lack support for packing transformations, which led to the development of GPAT. GPAT implements a region-based data layout transformation (packing) targeting loop nests found in dense tensor computations. It was implemented in MLIR Affine dialect, which specializes in this type of computations. The next chapter explores how the design principles of GPAT can be extended to support more general computations, such as those targeted by general-purpose compilers like LLVM.

Chapter 3

Region-Based Data Layout via Data Reuse Analysis

Reducing cache misses through data-layout transformation is a well-known and extensively studied problem. Cache-conscious data placement and data packing are two constrained definitions of this problem, both of which are NP-hard [53], [71]. This fundamentally hard problem is approached in general computations through heuristics that guide a set of transformation techniques, such as structure splitting and field reordering [15], [40], [90], [92], [93], [98], array regrouping [59], [93], [98], and pointer inlining [7], [86]. This set of techniques is named structure splicing [92]. For example, in structure splitting, the fields of existing structure types are split into two or more new types, as illustrated in Figure 3.1. Combined with a frequency-of-access analysis, structure splitting separates hot and cold fields, and with an affinity analysis, structure splitting groups fields that show high affinity. Once a structure type is split, all references in the program to that type and its memory allocation must be updated to reflect the new data layout. Frequency-of-access and affinity analyses are heuristics commonly used to guide such transformations.

Despite the extensive research on data-layout transformations, their utilization in production compilers is limited for three reasons. First, these transformations typically operate on a whole-program level and they try to achieve a globally-optimal data layout. In practice, the pattern of access to data structures varies in different regions of the program, meaning that there is no single global optimal layout. Therefore, the data layout selection must carefully trade off data-access improvements to different code regions. Second, the heuristics used for these transformations require

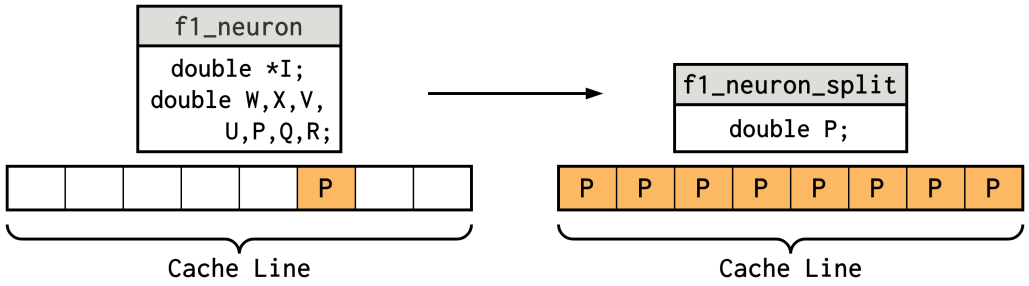


Figure 3.1: Type `f1_neuron` is split into two types, one containing only field `P`, and another omitted from the figure. The cache lines show the disposition of field `P` in an array of these structures.

a deep understanding of the program's runtime behaviour, such as relative access frequencies and affinity relations. Gaining this understanding requires expensive profiling techniques such as instrumentation and tracing which are impractical for most real-world applications. Third, a compiler must be able to prove that a transformation is legal before applying it. Many previous attempts at data-layout transformations cannot enforce legality – especially in type-unsafe languages such as C – and instead rely on manual intervention to verify correctness, which is impractical in a large-scale industry-setting deployment [59], [92], [93], [98].

Applying data structure splicing through region-based data-layout transformations is a central idea of this chapter. By focusing on regions of code, an analysis can statically determine the appropriate transformations, eliminating the overhead of monitoring the program's run-time behaviour. Ensuring legality is also significantly simpler with a region-based approach because a transformation only has to be legal for the target region. Once a transformation's legality is established, it can be applied automatically. The region-based approach was introduced by Mannarswamy *et al.* [63], Section 5.2 discusses the limitations of their work and highlights the differences in their approach to the problem. A crucial limitation in their work is the lack of data reuse analysis.

The region-based approach uses data copying to apply its transformations. To obtain performance improvement with this approach, it is not enough to identify regions that would benefit from a new data layout. The transformation's benefit must also overcome the copying overhead. This chapter presents RebaseDL, an analysis designed to identify profitable and legal opportunities to apply region-based

structure splitting and field reordering. To overcome the copying overhead, RebaseDL finds these opportunities within code regions that exhibit data reuse. Additionally, RebaseDL identifies transformation opportunities that do not involve structure types, that is, data packing opportunities. The main contributions of this chapter are as follows:

- RebaseDL, a static analysis that (i) combines data reuse with cache utilization to identify region-based data-layout transformation opportunities, and (ii) generalizes region-based data layout so it applies not only to data structures but also to non-aggregate types (Section 3.3).
- An implementation of this analysis in LLVM, a widely-used, modern compiler framework, enabling widespread use of RebaseDL (Section 3.3.3).
- An evaluation of the RebaseDL and its data-layout transformations, showing that many transformation opportunities are found in the SPEC CPU benchmark suites, demonstrating the challenges found in real-world applications and showing that speedups of up to 1.34x can be obtained with region-based transformations (Section 3.4).

3.1 Definitions and Notation

This chapter assumes the use of a Static Single-Assignment (SSA) IR. In this IR, loops are natural loops, composed of a subset of the CFG nodes [61]. A loop contains one header node that dominates all other nodes in the loop. IR instructions refer to memory addresses through opaque pointers, meaning that pointers are not explicitly typed. The type of the elements pointed by an opaque pointer is inferred by the uses of the pointer. Typically, accesses to a field of an aggregated data structure consist of adding an offset to a **base address**. In more complex data accesses, a sequence of offsets may be added to the base address, forming an **offset chain**.

Static array definitions and functions such as `malloc` allocate a consecutive **memory range** of addresses, given the input size of the range. Pointers referencing a memory range address are assumed to continue to reference the same memory range after offset operations. Still, distinguishing instructions that access separate memory ranges can be challenging. Such distinction is necessary for a precise analysis

that decides which memory elements would benefit from a new data layout and which would not. The minimal base address of a memory access, along with an alias analysis, is used to distinguish between memory ranges. The **minimal base address** is the base address used by the maximal offset chain of an access. A **maximal offset chain** is an offset chain that involves all discoverable offset operations applied to an address. The minimal base address is not necessarily the first byte of a memory allocation; it may be obtained from a load instruction or as a function argument. At compile time, a memory access is done through a pointer variable, and its minimal base address is a **minimal base pointer** variable.

A **candidate** for data-layout transformation is represented as a code region and memory range pair (L, M). In this pair, memory range M undergoes a data-layout transformation within the code region L. The target code regions consist of SESE loops. Target loops may even have multiple exits, as long as all exits but one lead to program termination, caused for example by assertions. These types of exits are represented by a special `unreachable` instruction. Loops are the target regions for transformation because they consume most of the computation time and often exhibit data reuse.

3.2 Data Layout Transformation

This chapter focuses on region-based structure-splitting and field-reordering transformations. The analysis detailed in Section 3.3 is designed to find candidates for these transformations. The steps to apply these transformations to a memory range M, which contains elements of a structure type T, and targets a loop L are the following:

i. Create a new structure type T' that contains only the fields from the memory range M accessed in loop L.

ii. Reorder the fields of type T' to match the order in which they are accessed in L.

iii. Immediately before the header node of loop L:
   (a) Allocate M', a new memory range that will hold elements of type T'.
   (b) Insert a copy loop that populates M' with the data of M that is accessed in loop L.

iv. Replace all instructions that access M by instructions that access the corresponding elements in M' within loop L.

v. Immediately after exiting loop L:

   (a) If there are writes to M', insert another copy loop to copy elements from M' back to M.

   (b) Deallocate M', if necessary.

The analysis is generalized to also discover candidates for a similar transformation where T is a non-aggregate type. In this case, the transformation is referred to as data packing [76]. Data packing involves nearly the same steps as the previous definition, with the exception that the first two steps are skipped because $T = T'$ for a non-aggregate type.

For structure splitting, the first step creates a new structure type containing all the fields utilized in the target loop. This is the splitting mode used in the evaluation of this chapter. The splitting mode could be changed to maximal splitting, which completely separates the fields of a structure. Maximal splitting thus transforms an array of structures into a structure of arrays. Moreover, the allocation of the new memory range may be done dynamically through a `malloc` call, or through an `alloca` call if the size of M' is a statically known constant. For stack allocations, the constant should be small enough to avoid stack overflow errors.

Running Example. The code snippet in Listing 3.1, extracted from the SPEC CPU 2000 benchmark `179.art`, serves as an example for the transformation. The variable `f1_layer` is a pointer to the structure type `f1_neuron`, which is also defined in the listing. Each loop is identified by its IV, *e.g.* $\texttt{loop}_{tj}$ is the loop defined in Line 6. In Listing 3.1 there are three distinct memory ranges, and the minimal base pointers of these memory ranges are `Y`, `f1_layer`, and `bus`. The target regions in this example are $\texttt{loop}_{tj}$ and $\texttt{loop}_{ti}$.

Listing 3.2 exemplifies the previously defined transformation applied to Listing 3.1, using a high-level language representation. The transformation was applied to the memory range referenced by the pointer `f1_layer` targeting $\texttt{loop}_{tj}$ in the original code. In Listing 3.2, the new structure type `f1_neuron_split`, containing only field `P`, is defined. The new memory range is allocated in Line 5 and the elements are

```
1  typedef struct {
2      double *I; double W; double X; double V;
3      double U; double P; double Q; double R;
4  } f1_neuron;
5
6  for (tj = 0; tj < numf2s; tj++) {
7    Y[tj].y = 0;
8    if (!Y[tj].reset)
9      for (ti = 0; ti < numf1s; ti++)
10       Y[tj].y += f1_layer[ti].P * bus[ti][tj];
11 }
```

Listing 3.1: Snippet of code from 179.art, part of SPEC 2000. For the running example, `numf2s` is 11 and `numf1s` is 10000.

copied in $\texttt{loop}_i$. Lastly, uses of the pointer `f1_layer` are replaced by the pointer `f1_layer_split` in $\texttt{loop}_{tj}$. In this example, there is no need to copy the transformed data back to the original memory range because there are no writes to the transformed data.

The contents of Listing 3.1 could be adapted so `f1_layer` becomes an array of doubles that contains all values of the `f1_neuron` type while maintaining the same computational semantics. By doing this, the access `f1_layer[ti]` would become `f1_layer[5+ti*8]`. The transformation applied in Listing 3.2 could be applied to this array version of Listing 3.1, now representing a data packing transformation instead of region-based structure splitting. The analysis in Section 3.3 can identify both these transformation candidates.

3.3 RebaseDL: The Analysis

The primary contribution of this research is RebaseDL, a static analysis that identifies candidates for the transformations mentioned in Section 3.2: region-based structure splitting, region-based field reordering, and data packing. RebaseDL faces several challenges. First, it needs to identify regions characterized by access patterns that cause low cache-line utilization. Candidates for region-based structure splitting are found in code regions that access a few fields of a particular data structure across multiple instances of the structure. Similarly, candidates for data packing are found in code regions that have strided accesses to arrays holding non-aggregate elements.

```
typedef struct {
  double P;
} f1_neuron_split;

f1_neuron_split *f1_layer_split = (f1_neuron_split *)
       malloc(numf1s * sizeof(f1_neuron_split));

for (int i = 0; i < numf1s; i++)
  f1_layer_split[i].P = f1_layer[i].P;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer_split[ti].P * bus[ti][tj];
}

free(f1_layer_split);
```

Listing 3.2: Region-based data layout applied to Listing 3.1

Second, it must determine whether applying a data layout transformation to these regions would indeed be beneficial. The main concern is the cost of the data copy, which introduces additional computational overhead. Third, as a static analysis, loop trip counts and array sizes are often dynamic, hindering the use of a precise analysis of cache usage in code regions. The next sections detail how these challenges were tackled.

3.3.1 Analysis Phases

RebaseDL operates in loop nests. For a loop nest, all its loops are potential target regions. For every loop L in the nest, and every memory range M accessed in the loop nest, the pair (L, M) constitutes a possible candidate. All possible candidates undergo six analysis phases. The first phase eliminates candidates that do not exhibit data reuse. The second phase finds candidates for which it is legal to apply data-layout transformations. The third phase checks if the candidate's accesses exhibit low cache-line utilization. The fourth phase eliminates candidates whose accesses to the memory range M may occur infrequently. The fifth phase estimates the cost-benefit ratio of using the transformed data layout in the target region and eliminates candidates that have a high cost relative to their benefit. The last phase

selects the candidates with the highest benefit, eliminating candidates that target overlapping code regions and the same memory range. The output of the analysis is the selection of candidates that remain after the last phase.

Before the first phase, the analysis must determine what are the distinct memory ranges accessed in the loop nest and which instructions access which memory ranges. Accesses that share the same minimal base pointer P_{min} are grouped because they access the same memory range. Similarly, accesses whose P_{min} variables must alias are grouped. If two pointers may alias, the transformation can later utilize code versioning to generate two code paths: one for the case where the pointers do not alias and another for when they do. A similar process is utilized by vectorization transformations.

Phase 1. Data Reuse. To mitigate the overhead introduced by the data copying in region-based transformations, this phase eliminates candidates that do not exhibit data reuse (temporal locality). In essence, this phase checks whether the accesses to the memory range M in loop L are invariant to loop L while still being variant to a child loop of L. If such invariance exists, loop L repeatedly accesses the same set of data elements in M, thus demonstrating data reuse.

The candidate ($\texttt{loop}_{tj}$, `f1_layer`) will be referenced as the **running example** from this point on. Consider the running example. It exhibits data reuse because the induction variable `tj` does not affect the indexing of the pointer `f1_layer`, only `ti` does. Therefore, `f1_layer` is invariant to $\texttt{loop}_{tj}$ and variant to $\texttt{loop}_{ti}$. After the transformation in Listing 3.2, the first iteration of the $\texttt{loop}_{tj}$ accesses all elements of `f1_layer_split`, then, the subsequent iterations access these elements again.

This invariance between accesses and loops is detected with a Scalar Evolution (SCEV) analysis [4], [72], which can be queried about how variables evolve in a loop. The response to such queries yields a SCEV expression that may contain a recurrence in the format {`starting value, operation, step`}. For example, the recurrence that describes the evolution of the IV of $\texttt{loop}_i$, in Listing 3.2 is {`0,+,1`}.

For each candidate (L, M), this phase inspects the SCEV of all pointers to the memory range M used in loop L. These pointers must be invariant to loop L and variant to a child loop of L. To be invariant to a loop l, a SCEV expression must

meet two conditions: (i) it cannot contain a recurrence generated by loop l, and (ii) it cannot involve any variables whose definition is contained in loop l. The latter condition can be relaxed if the variable defined in loop l is the result of a load instruction whose pointer operand has a SCEV that meets the two previous conditions, meaning that the load is inside of loop l but it is invariant l. These constraints represent a conservative notion of reuse that may not capture all cases of data reuse and is similar to previous work [76]. This analysis allows for independent future improvements in each of its phases.

Phase 2. Legality. Ensuring that a data transformation is legal is a limitation in many previous attempts to perform data-layout transformations, particularly in type-unsafe languages like C/C++. Hundt *et al.* [40] described many requirements for their whole-program data-layout transformations to be legal in such languages. Their experiments exemplify that many opportunities are lost due to non-conformity with their requirements. In region-based transformations, the legality requirements only need to be met at target regions, not globally. In this phase, a candidate (L, M) that does not meet the requirements in loop L is eliminated.

The legality requirements of a candidate targeting loop L and memory range M are the following:

Intraprocedural: For simplicity, the analysis is limited to the scope of a procedure, and thus candidates must not require an interprocedural analysis. Such a candidate is identified by inspecting the pointer arguments of call instructions within loop L. If any of these arguments is a pointer to M, the candidate is eliminated.

No Global References: If the minimal base pointer P_{min} of M is a global variable, the target region L must not have a call to a function that references P_{min} (in itself or by its calls).

Known Loop Bounds: Loops nested inside loop L must have bounds that are known, and the SCEV of these bounds must be invariant to loop L and its child loops. Otherwise creating the copy loops would not be possible. An exception to this rule applies for loops l where accesses to M are invariant to l.

Single Minimal Base Pointer: A pointer may have multiple minimal base pointers

P_{min} due to control flow. Without going through any `phi` instructions, all pointers to M in loop L must share the same P_{min}. Additionally, pointer P_{min} must have a SCEV that is invariant to loop L and its child loops.

Phase 3. Cache Utilization. This phase estimates the cache-line utilization ratio of a given candidate (L, M), defined in (3.1). In this equation $Bytes_{acc}$ is the number of bytes that are intentionally accessed by the program from memory range M in loop L. $CacheLines_{acc}$ corresponds to the number of cache lines accessed due to these intentional accesses, and CLS is the cache line size in bytes.

$$UtilizationRatio = \frac{Bytes_{acc}}{CacheLines_{acc} \times CLS} \tag{3.1}$$

A low cache-utilization ratio suggests that the majority of the bytes loaded into cache lines associated with memory range M are unused. If the ratio is above a threshold, the candidate is eliminated as the transformation will likely not be beneficial. This threshold can be set based on a sensitivity analysis. For the evaluation, this threshold is set to 0.5, a more conservative choice than the 0.7 threshold used in Mannarswamy's RBSL [63].

Consider the example in Listing 3.1, assume `numf2s` equals 11, `numf1s` equals 10000, and a `double` data type occupies 8 bytes in memory. Under these conditions, the running example has $Bytes_{acc}$ equal to 80000 bytes. The offset between the field `P` of consecutive instances of the struct type `f1_neuron` in memory is 64 bytes. Given that the cache line size (CLS) is 64 bytes, each access to `P` in Line 10 fetches a cache line of 64 bytes but only uses 8 bytes. Thus, $CacheLines_{acc}$ is equal to 10000, yielding a utilization ratio of 0.125.

Phase 4. Relative Access Frequency. So far, no restrictions have been placed on the presence of control flow divergence within the CFG of a candidate's target loop. A worst-case scenario occurs when the accesses to the transformed memory range M' only execute if a condition is true, and this condition is rarely true. In Listing 3.1 this situation could happen due to the if condition in Line 8. The main problem is that after the transformation, the allocation and copy loop execute regardless of the outcome of the if condition. Moreover, as a static analysis, the compiler does not

know the dynamic behaviour of the program, unless provided with feedback from a prior execution.

One approach is to eliminate candidates (L, M) that may not execute the load and store instructions to memory range M in loop L due to control-flow divergence. This approach is too restrictive for the benchmarks tested (SPEC CPU), as it would eliminate nearly all candidates found. Instead, RebaseDL uses a static Basic Block Frequency Analysis [60]. This analysis provides an estimate of the execution frequency of a basic block, relative to the entry block of a function. RebaseDL uses this analysis to calculate *AccessFreqRatio*, defined in (3.2) and (3.3), with function *Freq* returning the execution frequency of a basic block. The component *header*(L) returns the header block of loop L. B_{dom} represents a set of basic blocks composed of the first dominator of the loads and stores accessing M within loop L that have L as their immediate parent loop. If this ratio is below a threshold, the candidate is eliminated, as it is unlikely that the accesses to the transformed data layout are executed frequently in loop L. This threshold is set to 0.35, and a sensitivity analysis for this threshold is left for future work.

$$Freq_{min}(B_{dom}) = min(Freq(b)), \forall b \in B_{dom} \tag{3.2}$$

$$AccessFreqRatio = \frac{Freq_{min}(B_{dom})}{Freq(header(L))} \tag{3.3}$$

For example, in Figure 3.2 there are two loops in the CFG: one composed of all nodes whose header is node *Loop*, and another composed of the single node *Innerloop*. Assume a candidate (L, M), where L is the outer loop of the CFG, and there is a load to M inside the *Innerloop* block. Following the dominator tree of this CFG in Figure 3.2, it is clear that the block *Then* is the first dominator of *Innerloop* that has the outer loop as its immediate parent. Thus, *AccessFreqRatio* equals *Freq(Then)* divided by *Freq(Loop)*.

Phase 5. Cost-Benefit. Leveraging the terms $CacheLines_{acc}$ and $Bytes_{acc}$ introduced in equation (3.1), this phase computes the cost-benefit ratio of applying RebaseDL's transformation to a candidate. RebaseDL uses the number of unused

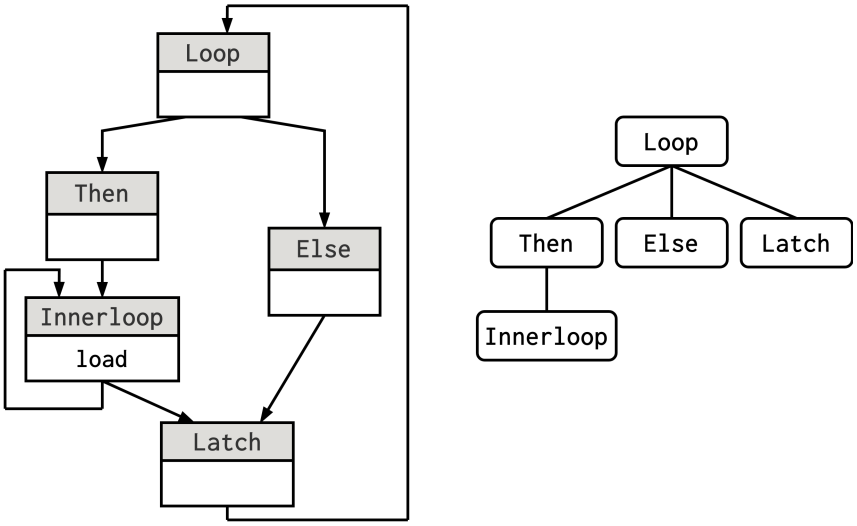


Figure 3.2: CFG of a loop nest and its dominator tree.

bytes loaded to the cache as a key metric in determining if the transformation is beneficial to apply. This metric is obtained by (3.4). For the running example, *UnusedBytes* is calculated as $(10000 \times 64) - 80000 = 560000$.

$$UnusedBytes = (CacheLines_{acc} \times CLS) - Bytes_{acc} \tag{3.4}$$

The data-layout transformation selectively copies the data that will be used in the target region. Assuming that the allocated data starts at the beginning of a cache line and that there is no padding in the transformed data layout, all unused bytes are eliminated, except for a possible tail at the last cache line that may not be fully used. The number of cache lines required to store the copied data for the transformation, *CacheLines_trn*, is given by (3.5). For the running example, $CacheLines_{trn}$ is equal to 1250.

$$CacheLines_{trn} = \left\lceil \frac{Bytes_{acc}}{CLS} \right\rceil \tag{3.5}$$

The number of unused bytes loaded to the cache after applying the transformation, *UnusedBytes_trn*, is defined by (3.6). For the running example, $UnusedBytes_{trn}$ equals 0.

$$UnusedBytes_{trn} = (CacheLines_{trn} \times CLS) - Bytes_{acc} \tag{3.6}$$

The benefit of the transformation is the reduction in unused bytes loaded into the cache. The first phase of the analysis guarantees that the candidate's memory-range accesses remain invariant to the target loop. As a consequence, this benefit is multiplied by the trip count of the target loop $TripCount_L$. The resulting benefit is given by (3.7). The benefit of the running example is $560000 \times 11 = 6160000$

$$Benefit = (UnusedBytes - UnusedBytes_{trn}) \times TripCount_L \tag{3.7}$$

The cost is given by the number of cache lines loaded to transform the data in the copy loop, multiplied by two if the transformation requires a second copy loop to store modified elements back to the original memory range. The cost is calculated by (3.8). The cost for the running example is 64000 because `f1_layer` is only read.

$$Cost = \begin{cases} CacheLines_{acc} \times CLS, & \text{if } M \text{ is read only in } L \\ CacheLines_{acc} \times CLS \times 2, & \text{otherwise} \end{cases} \tag{3.8}$$

Finally, the cost-benefit ratio of a candidate is given by *Cost/Benefit*, which is equal to 0.01 for the running example. This phase eliminates candidates that do not have a cost-benefit ratio smaller than 0.5 to eliminate candidates with a cost higher than the benefit and to ensure that candidates that have a $TripCount_L$ that is too small are also eliminated. A sensitivity analysis for this threshold is left for future work. At this point, the remaining candidates are deemed beneficial for the transformation defined in Section 3.1.

Phase 6. Greedy Selection. The last phase of the RebaseDL selects between conflicting candidates. Candidates conflict if they target overlapping code regions and transform data in the same memory range. In this phase, candidates are sorted from best to worst based on the computed *Cost/Benefit* values from the previous phase. If there is a tie, the loop depth is the deciding factor. Lower loop depths are considered better because candidates that target a loop at lower depths impact a broader region. Lower loop depths are also preferred because RebaseDL allows the outermost loop to be a target code region. Targeting the outermost loop enables the potential expansion of the region to encompass neighboring loop nests if they refer to the same memory ranges. Region expansion and fusion, however, are outside

the scope of this chapter. Once candidates are sorted, they are selected one by one, excluding any conflicting candidates.

3.3.2 Underneath the Analysis

To compute the cache utilization and cost-benefit relation, RebaseDL needs to obtain estimates for the following terms:

***TripCount*$_{\mathbf{l}}$.** The estimated trip count of a loop l. If the loop is computable, it is equal to the trip count of the loop. Otherwise, the accesses in loop l serve as hints for an estimate. If an offset operation indexes an array of fixed size and its SCEV contains a recurrence created by loop l, the trip count of l is estimated to be the size of the array. If there is no indexing to a fixed-sized array, the trip count defaults to a value, which is set to 10. While 10 is an arbitrary value, unknown trip counts default to a small value to avoid inflating the *Benefit* term defined in Phase 5, which is multiplied by a trip count. This estimation can be improved with a Feedback-Directed Optimization (FDO) framework.

***Bytes*$_{\mathbf{acc}}$.** The number of distinct bytes accessed in the memory range M within loop L, for a candidate (L, M). The value $Bytes_{acc}$ is estimated by summing the number of bytes accessed by all pointers to M in L, and multiplying this number by the trip count of all loops contained in L that change the offsets applied to these pointers. The amount of bytes accessed by a pointer p is inferred from the type of the resulting value of the uses of p in load and store instructions. A loop l changes the offsets applied to a pointer if the SCEV of the pointer is not invariant to l. The conditions for a SCEV to be invariant to a loop are defined in Phase 1 of the analysis. For the running example in Listing 3.1, the pointer `f1_layer` is used in a load instruction to load the value `P` of type double, thus accessing 8 bytes. This instruction is inside `loop`$_{ti}$ and `loop`$_{ti}$, however, the offset applied to `f1_layer` only varies with `loop`$_{ti}$, which has a trip count of 10000. Therefore, $Bytes_{acc}$ for this example is 80000.

***CacheLines*$_{\mathbf{acc}}$.** The number of distinct cache lines accessed due to the byte accesses accounted for in $Bytes_{acc}$. This term requires splitting the offset applied by a pointer's offset chain in two: constant offset and loop offset. The **constant offset** is the sum of all constant offsets applied to a pointer, and if any offset operation involves a variable, its offset is considered to be zero. The **loop offset** is the difference in offsets applied to a pointer between two consecutive iterations of a loop l. For example, the pointer to `f1_layer` used to load the variable `P` in Listing 3.1 has a constant offset of 40 bytes – which is the offset from the start of the struct `f1_neuron` to field `P`. The loop offset of this instruction is 0 regarding $\texttt{loop}_{tj}$, whereas for $\texttt{loop}_{ti}$, the loop offset is 64 bytes due to the indexing of the IV `ti`.

$CacheLines_{acc}$ is obtained by mapping the $Bytes_{acc}$ of a candidate (L, M) to cache lines and counting the total of cache lines mapped, assuming that an unlimited number of cache lines are available. Given a pointer p to memory range M used by load and store instructions in loop L, the analysis defines: (i) the number of bytes it accesses, (ii) its constant offset, (iii) its loop offset relative to all loops in L, (iv) and the trip count of these loops. With these values defined, the analysis steps through every iteration of the loop nest starting in loop L. For each iteration, the analysis applies the constant offset of p, and the respective loop offset given by the loops in L to obtain the cache lines being accessed. This mapping process is done for all pointers p. $CacheLines_{acc}$ equals the total of cache lines mapped. The bottom left of Figure 3.1 shows an example of this mapping for a single iteration of $\texttt{loop}_{ti}$ for pointer `f1_layer` in Listing 3.1. The mapping of this example would be this cache line replicated 10000 times. Although not an exact simulation of a real cache, this method gives a good sense of cache utilization and unused bytes in cache lines, which is what the analysis effectively requires.

3.3.3 RebaseDL in LLVM

RebaseDL was implemented in the LLVM compiler infrastructure project [50] as an out-of-tree pass. This pass works as a compile-time pass that can be inserted before vectorization passes and target-specific optimizations. However, during link time, the compiler can inline functions between different compilation modules, and all function definitions are available. Therefore, RebaseDL is run as a link-time pass

to enable a higher-precision analysis and to avoid problems with the legality checks when they encounter a function whose definition is unknown. The implementation is available as part of an artifact [75].

At the time of writing, the implementation comprises the analysis component described in Section 3.3.1. This analysis leverages the following analyses in LLVM: LoopInfo, ScalarEvolution, AliasAnalysis, BlockFrequencyInfo, and DominatorTree. RebaseDL's transformation described at the end of Section 3.1 is yet to be implemented in future work. For the following evaluation, the transformation was manually applied to the candidates identified by the analysis. Debugging information was used to display the origin of relevant IR instructions in the high-level C/C++ code.

3.4 RebaseDL Evaluation

This section evaluates RebaseDL and whether or not its analysis combined with a data-layout transformation can improve the performance of real-world applications. In particular, it addresses the following questions: ① Can RebaseDL identify candidates in the SPEC CPU benchmark suites? ② Does data reuse impact the performance of region-based data layout transformations? ③ How does the transformation affect the performance of the target regions? ④ Does the change in data layout improve cache utilization?

3.4.1 Setup

The machine used in the evaluation runs Ubuntu 20.04.6 (Kernel 5.4.0-148) and is equipped with the hardware specifications defined in Table 3.1. The benchmarks used for the evaluation are the C/C++ benchmarks from SPEC CPU 2000[1], 2006[2], and 2017[3] — benchmarks commonly used in previous works on data-layout transformations [40], [63], [92], [93]. For SPEC CPU 2017, the *SPECrate* variant of the benchmarks was utilized. For the SPEC CPU 2006 and 2000, small patches were added to some benchmarks to avoid compilation errors. These patches are

[1]Version 1.2
[2]Version 1.2
[3]Version 1.1.0

Table 3.1: Experimental machine configuration

CPU	AMD Opteron Processor 6308
Frequency	2.6 GHz (locked)
L1 dCache	4 × 16 KiB, 4-way, 64 B line
L2 Cache	2 × 2 MiB, 16-way, 64 B line
L3 Cache	2 × 8 MiB, 48-way, 64 B line
Memory	64 GiB DDR3

Table 3.2: The number of candidates found by RebaseDL in SPEC CPU benchmarks. In total, 71 candidates were found.

Benchmark	**Candidates**	**Benchmark**	**Candidates**
175.vpr	1	502.gcc_r	3
179.art	2	510.parest_r	42
445.gobmk	1	525.x264_r	1
447.dealII	18	526.blender_r	1
464.h264ref	1	538.imagick_r	1

documented in the artifact. All SPEC benchmarks use the *ref* set of inputs in the experiments.

The benchmarks were compiled with Clang and linked with LLVM's linker LLD, both from LLVM version 16.0.0. The files used to compile SPEC are available in the artifact. But notably, the following compilation and linking flags were utilized: `-fno-unroll-loops -O3`, `-march=native`, and `-flto`. The first flag disables loop unrolling because it affects the analysis' ability to find candidates in its current implementation when run at link time.

Answering ①, RebaseDL finds many opportunities in the SPEC CPU benchmark suites. Table 3.2 shows which benchmarks contain candidates and the number of candidates found. These results indicate that the code patterns of data reuse and poor cache utilization are present in real-world applications. However, inspection using Perf [31] reveals that many candidates are in code regions that are not frequently executed, and some candidates are not executed with SPEC's inputs. The exception is the *179.art* benchmark, where the target regions of the two candidates identified by RebaseDL are in frequently executed code.

3.4.2 Experimental Methodology

The evaluation discusses two experiments. The first (Section 3.4.3) uses the Google Benchmark tool [32][4] to collect an average of 5000 executions for each data point presented. The second experiment (Section 3.4.4) uses the PAPI library [80][5] to obtain performance metrics of the candidates' target regions. In the second experiment, each benchmark was run 20 times, and performance metrics were collected per region. Note that the same region of code can be executed multiple times in a single run. Each run aggregates the total value of a metric per region (*e.g.*, total execution time spent in a region). The results show the average of these totals across the 20 runs and an error bar that represents the 95% confidence interval around the mean. For some of the regions that do not execute with the inputs provided by SPEC, slight modifications were added to the benchmark to make the region execute. For example, the region of the candidate found in *175.vpr* does not execute with the SPEC inputs, however, by commenting one if condition in the main function of the benchmark, the region executes without affecting the correctness of the execution. Such modifications are recorded in the artifact.

Two configurations were evaluated in both experiments: (i) **Clang**, in which a benchmark is compiled with Clang and linked with LLD, using the flags mentioned in Section 3.4.1, and (ii) **RebaseDL**, the data-layout transformation defined in Section 3.2 (following step *i.* to *v.*) manually applied to the previous configuration. Because of the considerable effort of applying transformations manually, not all candidates were transformed. In particular, *510.parest_r* was not used in the evaluation due to its large volume of candidates and its similarity with the *447.dealII*. Candidates whose region does not execute are not shown in the figures and tables.

3.4.3 Data Reuse Impact

This experiment demonstrates the impact that data reuse has on the performance gains of region-based transformations to answer ②. For this, the region that obtains the best performance improvement in Section 3.4.4 – region *trainmatch* of *179.art* – was recreated outside of SPEC CPU 2000. This region is composed of the code

[4]Version 1.8.2

[5]Commit hash e8d7767

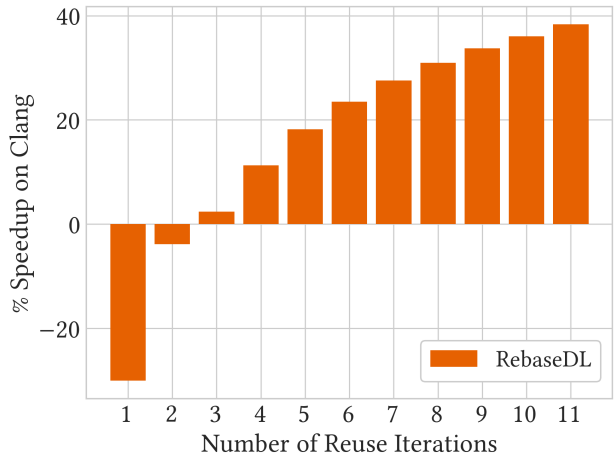


Figure 3.3: Speedup of RebaseDL over Clang while varying the number of reuse iterations for the region *trainmatch* of *179.art*.

shown in Listing 3.1. The sizes of dynamic arrays and trip counts correspond to those found when using the SPEC *ref* input, and arrays are initialized with random values. The value of the upper bound of $\texttt{loop}_{tj}$ (variable `numf2s`) is either 10 or 11 for this region. Figure 3.3 shows the speedup (or slowdown) obtained by the RebaseDL configuration transforming the candidate ($\texttt{loop}_{tj}$, `f1_layer`) over Clang when the value of `numf2s` is changed from 1 to 11. Changing this value effectively changes how many iterations will reuse the transformed data. The worst performance is obtained when `numf2s` equals 1, that is, when there is no data reuse, making the copy loop added by the transformation mostly overhead without benefit. As the number of reuse iterations increases, performance gains also increase.

3.4.4 Transformation Evaluation

This experiment shows the performance of region-based data-layout transformations on SPEC CPU benchmarks to answer questions ③ and ④. Figure 3.4 shows the results for the benchmarks *175.vpr*, *179.art*, and *445.gobmk*. This graph shows the execution time of each region that was transformed, where the regions are identified by names on the x-axis. The results on these benchmarks show that the RebaseDL configuration can improve performance, and the performance counters for these regions also show that cache utilization is improved. The thresholds employed by the analysis could all be set to more conservative values and still find these candidates.

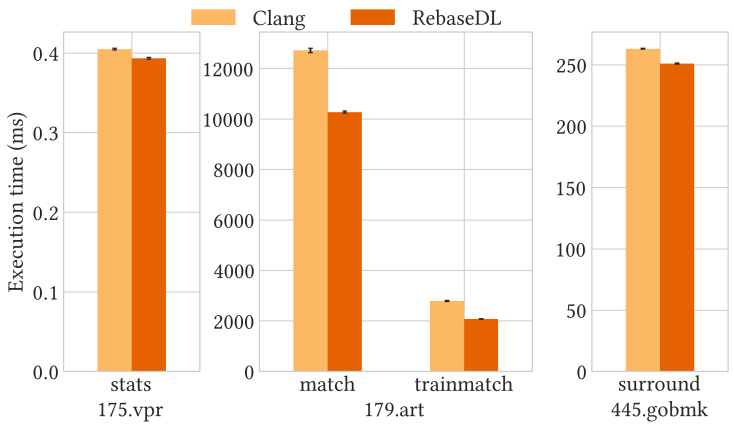


Figure 3.4: Execution time before and after the transformation for code regions in the benchmarks *175.vpr*, *179.art*, and *445.gobmk*. From left to right, RebaseDL obtains the following speedups over Clang: 1.03x, 1.24x, 1.34x, 1.05x.

The results for the remaining benchmarks are shown in Table 3.3 and Table 3.4 as they have differences in execution time that are more subtle. Most of the results of Table 3.4 show a slight degradation in performance. The factors that caused it are discussed in this section. Among them, the table highlights the dynamic trip count of the target loop of the candidates (Reuse Iterations).

For *175.vpr*, the analysis identifies a cache utilization ratio of 0.208. The transformation performs structure splitting of a single field, obtaining a slight speedup of 1.03x, and an improvement of the same order is seen in L1 and L2 data-cache misses. Another point to note is that there is a more drastic reduction in the dynamic number of instructions of 1.17x. This reduction is due to an index computation done inside a double-nested loop that requires a pointer indirection, after the transformation, this computation is only done in the copy loop outside of the loop nest.

The benchmark *179.art* shows the best results overall with speedups of 1.24x and 1.34x for the regions *match* and *trainmatch*, respectively. Regarding the execution time of the entire benchmark, RebaseDL obtains a speedup of 1.04x. The two transformed regions are very similar to Listing 3.1, however, *trainmatch* obtains a better speedup because the if condition in Line 8 is always true for this region. These regions present a cache utilization ratio of 0.125. In this example, the transformation reduces L1 data-cache misses by 1.7x, and reduces L2 data-cache misses by 1.3x and 1.7x for the regions *match* and *trainmatch*, respectively.

Table 3.3: Execution time of Clang and RebaseDL for code regions of SPEC benchmarks not shown in Figure 3.4. The ± values represent the 95% confidence interval around the mean execution times reported.

		Clang	RebaseDL
	Region	**Execution time (ms)**	**Execution time (ms)**
447.dealII	estimate2	7892.03 ± 18.22	7945.52 ± 12.3
	estimate3	2161.01 ± 11.73	2191.46 ± 7.13
	estimate4	2094.93 ± 8.33	2100.48 ± 6.30
	estimate5	651.33 ± 4.65	670.28 ± 2.90
	estimate6	653.57 ± 3.87	668.61 ± 2.33
	estimate7	641.21 ± 3.11	648.06 ± 1.88
	fe_q1	110.18 ± 0.09	108.96 ± 0.06
464.h264ref	macroblock	2747.48 ± 8.67	2747.54 ± 10.83
502.gcc_r	dominators	69.09 ± 0.15	69.66 ± 0.21
	ira_init	0.0422 ± 0.0003	0.0456 ± 0.0002
	omega	3.954 ± 0.029	4.050 ± 0.026
525.x264_r	encoder	662.78 ± 2.00	664.58 ± 2.36

The analysis identifies a candidate in *445.gobmk* that has a very low cache utilization of 0.062. The transformation applies structure splitting to a single field and statically allocates the new memory range because the original memory range was also statically allocated. After the transformation, there is a speedup of 1.05x to the region, which comes from a reduction of 1.4x in L1 data-cache misses.

The results for *447.dealII* are in Table 3.4. Among its candidates, the compiler could not gather enough debugging information for two regions, and one region requires a transformation that modifies code that is not part of the SPEC benchmark. These candidates were not transformed. The *estimateX* regions are a repeating pattern of code, and they share the same cache utilization ratio of 0.312. In these regions, the transformation splits the `std::vector` class that contains three pointers fields and copies only the one pointer that is used in the region. The slight performance degradation of these regions is due to the low trip count of the target loop, which is unknown at compile time and is shown in Table 3.4. This means that the data reuse expected by the compiler does not happen in practice for these regions. The remaining region *fe_q1* has a cache utilization ratio of 0.438, and it obtains a slight speedup of 1.01x after two fields from the `FullMatrix` type are split into a new type.

In *464.h264ref*, RebaseDL identifies a cache utilization ratio of 0.250 for a

Table 3.4: Speedup and reuse iterations for code regions of SPEC benchmarks not shown in Figure 3.4.

	Region	Speedup over Clang	Reuse Iterations
447.dealII	estimate2	0.993	1
	estimate3	0.986	1
	estimate4	0.997	1
	estimate5	0.972	1
	estimate6	0.978	1
	estimate7	0.989	1
	fe_q1	1.011	125, 27, or 8
464.h264ref	macroblock	1.000	2
502.gcc_r	dominators	0.992	2
	ira_init	0.926	2
	omega	0.976	1 or 2
525.x264_r	encoder	0.997	0

candidate in a nested loop with 6 levels, where the target loop is at depth 4. The target memory range is a static array of 17 elements of the `PixelPos` type. The transformation splits two fields of this type into a static array of 4 elements, as only 4 out of the 17 elements are accessed in the target loop. For this benchmark, the transformation shows no performance difference within the confidence interval, which is likely due to the small size of the memory range that can fit in a few cache lines.

For *502.gcc_r*, three candidates were identified. The trip count of the target loop for these regions is unknown at compile time but evaluates to very low values in practice, which are shown in Table 3.4. On the *dominators* region, a candidate with a cache utilization of 0.25 is identified in a triple-nested loop, in which the outermost loop is the target region. The transformation performs splitting of a single field. Similarly to *175.vpr*, the copy loop in this region uses an index computation that requires a pointer indirection, but unlike in *175.vpr*, this index computation cannot be eliminated from the original loop because it is still needed after transformation in the target loop. Therefore, the copy loop has an additional overhead of duplicating this index computation. On the *ira_init* region, which runs very briefly, a candidate that has a cache utilization of 0.236 is found. This is the first example of data packing instead of structure splitting, as the original memory range is a 3-dimensional array of constant size $87 \times 27 \times 2$ that is packed into a matrix of size 87×2. This region has

an enclosing if condition that evaluates to false with a probability of 50%, however, the copy loop executes regardless of the outcome. Lastly, the region *omega_problem* splits a single field in a region with a cache utilization ratio of 0.312. However, the innermost loop of the region has a trip count that is either 0 or 1 in practice, meaning that at most one element is accessed in the region. The low data reuse, combined with the other factors mentioned lead to slight slowdowns for these regions.

The last benchmark *525.x264_r*, RebaseDL identifies one candidate with a cache utilization ratio of 0.281 and the transformation splits 2 fields. The target loop, however, has a trip count that evaluates to 0 for the SPEC *ref* input. As such, the region obtains a very slight decrease in performance.

For all benchmarks shown in Table 3.4 that see a decrease in performance, the low number of (dynamic) reuse iterations is one of the causes. This fact highlights the importance of a reuse analysis for region-based transformation, which was not present in previous works [63]. In a production implementation of the analysis and code transformation, guards can be introduced to select alternative code paths when the trip counts are very low.

3.5 Discussion

This chapter presents RebaseDL, a static, link-time analysis that identifies region-based data-layout transformation opportunities. Specifically, it targets opportunities for structure splitting, field reordering, and data packing. RebaseDL utilizes a region-based approach, enabling simpler legality checks and avoiding runtime-based heuristics to define which data-layout transformations to apply. Opportunities are identified through cache utilization along with a data reuse analysis, a component that is absent in previous works. This chapter demonstrates that many transformation opportunities are identified in the SPEC CPU benchmark suites. Applying the transformations can result in speedups of up to 1.34x within the transformed region. However, the evaluation highlights challenges associated with unknown loop trip counts, which can cause slight decreases in performance. In addition, the inaccurate alias analysis present in LLVM is a challenge for performing the proposed transformation automatically. These challenges can be addressed by adding guards

to the generated code to select a code path that does not use the transformation based on trip count values and aliasing of the involved pointers.

With the challenges of optimizing general-purpose code in mind, the research presented in the next chapter shifts focus to the Artificial Intelligence (AI) domain and the application of data layout transformations within it. A core component of many AI models is the convolutional layer, which is often optimized using a data layout transformation known as Im2col. This transformation enables the use of GEMM kernels, which in turn rely on data packing for high performance. Prior work has aimed to reduce the overhead of Im2col while preserving its benefits, but a closer examination reveals that existing approaches do not exploit GEMM kernels and their packing strategies to fully eliminate the need for Im2col. The next chapter introduces a novel convolution algorithm that achieves high performance by leveraging the implicit region-based data layout transformations performed within the GEMM kernel.

Chapter 4

Zero-Copy GEMM-Based Fast Convolution

Convolutional layers are the core computational component of CNNs, accounting for the majority of CNN execution time. Thus, much research has focused on optimizing the performance of these layers. Evidenced by hybrid convolution algorithms employed by modern deep learning frameworks, no single method is optimal for all convolutions. Each approach presents trade-offs in terms of performance, generality, and portability. This complexity comes from: (i) the wide range of parameters that modify convolution behavior (as listed in Table 4.1); (ii) the tensor layouts of input and output tensors, which are often pre-defined by the framework in use; (iii) the diversity of execution targets. While this chapter focuses on CPU execution, even within this domain, supporting a wide variety of CPU microarchitectures remains a significant challenge. The recent inclusion of matrix multiplication units in some CPUs offers an opportunity for better performing implementations but further complicates code generation and optimization [8].

Common approaches to optimizing convolution can be categorized into direct convolution and those based on GEMM, Fast Fourier Transform (FFT), and Winograd's minimal filtering algorithm. FFT-based convolution transforms the operation into the frequency domain, where convolution becomes element-wise multiplication, however, its performance is sensitive to small filter sizes, whereas Winograd-based methods reduce the number of arithmetic operations required but often introduce numerical instability [54], [99]. Direct convolution approaches aim to optimize the naive implementation by applying loop tiling, loop reordering, and custom memory

layouts, among other techniques, to improve data reuse and better utilize computation units [29], [94], [97]. These techniques can be tailored through auto-tuning strategies or analytical performance models akin to those used in high-performance GEMM implementations. However, auto-tuning is computationally expensive, and designing analytical models that generalize well across different microarchitectures is both time-consuming and challenging. GEMM-based methods reduce convolution to matrix multiplication, traditionally using the Im2col transformation [10]. This approach is popular because it leverages the extensive engineering efforts behind optimized GEMM kernels, which are widely available across hardware platforms through Basic Linear Algebra Subroutines (BLAS) libraries [33], [81], [82]. Its primary drawbacks are the additional memory overhead and transformation cost of Im2col. Recent research efforts try to reduce Im2col's overhead, but they typically support only a subset of convolution configurations and do not fully eliminate Im2col's memory overhead [2], [16], [47], [69]. An alternative is to fuse the Im2col transformation directly into the GEMM kernel [24], [77], [83], [96]. While this approach can effectively eliminate the explicit transformation and achieve strong performance, it requires modifying the internals of the GEMM implementation. As a result, such methods lack portability across different BLAS libraries and hardware architectures.

This chapter introduces the Zero-Copy Convolution (ZConv), a novel GEMM-based convolution algorithm that fully eliminates the data copies required by Im2col-based approaches, while relying on unmodified GEMM kernels. ZConv is a simple, high-performance, portable, and fully parallelizable algorithm that supports a wide range of convolution configurations. Its core insight is that convolution can be decomposed into multiple GEMM calls that directly access the image tensor by exploiting the dimension and stride parameters accepted by BLAS libraries. By utilizing unmodified GEMM kernels and using conventional input tensor layouts, ZConv can leverage multiple BLAS libraries and is easily integrated into existing deep learning frameworks. ZConv's algorithm delegates optimizations to the BLAS library and is fully fused into a single loop nest. Its two outermost loops have no dependencies between iterations, and thus can be trivially parallelized. Additionally, a low-memory-overhead extension of ZConv is proposed to support dilated and grouped

convolutions. Together, these algorithms cover the full range of configurations supported by PyTorch's *Conv2D* operator, except for padding mode, which is assumed to be zero-padding [3].

Performance evaluation with a large dataset introduced by Alvarenga *et al.* [1], which includes over 9000 unique convolutional layers from more than 1200 vision models, reveals that ZConv outperforms both prior work and the state-of-the-art convolution implementations used by PyTorch. This large-scale evaluation enabled the creation of simple heuristics based on convolution parameters to decide when to utilize ZConv over alternative methods to maximize model performance. The integration of ZConv into PyTorch's hybrid convolution implementation for end-to-end model evaluation results in inference speedups of up to 1.8× over unmodified PyTorch.

In summary, this chapter makes the following contributions:

(i) ZConv, a novel GEMM-based convolution algorithm that requires zero data copies and utilizes unmodified GEMM kernels (Section 4.2.1).

(ii) An extended version of ZConv that supports dilated and grouped convolutions with low memory overhead (Section 4.2.2).

(iii) An extensive evaluation of ZConv against prior and state-of-the-art methods, highlighting the convolution configurations where it outperforms alternative methods (Section 4.3.2).

(iv) A description of how to integrate ZConv into PyTorch and its hybrid convolution implementation (Sections 4.2.3 and 4.3.2).

(v) An end-to-end evaluation of ZConv integrated into PyTorch by running inference across a large collection of vision models (Section 4.3.3).

4.1 Convolution and GEMM

Convolution is an operation that applies a filtering mask over an input tensor, usually an image [42]. This operation was incorporated into neural networks via convolutional layers, where the filtering weights are learned during training [55]. CNNs are designed to extract relevant features from images using convolution as a fundamental building block [56]. Despite the rise of transformer-based models, CNNs

Table 4.1: Notation for convolution parameters and input tensor dimensions.

N:	Batch	OH:	Output height	FH:	Filter height
IH:	Image height	OW:	Output width	FW:	Filter width
IW:	Image width	OC:	Output channels	SH:	Stride height
IC:	Image channels	GR:	Groups	SW:	Stride width
DH:	Dilation height	DW:	Dilation width	PH:	Padding height
PW:	Padding width				

such as MobileNet [39] and ResNet [37] remain widely used. Convolutional layers are also employed in some vision transformer architectures. For brevity, this chapter refers to convolutional layers as convolution, targeting two-dimensional convolution throughout.

A naive implementation of convolution is shown in Listing 4.1, written in C with simplified tensor indexing for readability. This implementation takes three input tensors: the input image, filters, and output (illustrated in Figure 4.2a), along with an optional bias array. These four-dimensional tensors can have different data layouts, that is, the order in which their dimensions are stored in memory. Line 2 specifies the layouts used in this implementation. In addition to the input tensors, the implementation utilizes several parameters defined in Table 4.1, which modify the convolution's behavior. Padding parameters, however, are omitted in Listing 4.1 and subsequent listings for brevity. To illustrate the naive execution, Figure 4.1 expands the iterations over the output height and width loops, corresponding to Lines 12 and 13 of Listing 4.1. The highlighted squares indicate the elements accessed in each iteration. In the image tensor, the groups of highlighted elements are referred to as patches. At the center of the figure, the element-wise multiplications followed by a sum reduction represent the computation performed inside the loop in Line 18.

The naive implementation performs poorly for most convolution parameter configurations. To address this, general matrix-matrix multiplication (GEMM)-based implementations are commonly used to optimize convolution, as the operation can be reduced to GEMM via the Im2col transformation [10]. Furthermore, convolutional layers with a filter height and width of one, known as pointwise convolutions, can be reduced directly to GEMM without requiring any transformation. This approach was popularized by the Caffe framework [43] and remains in use in modern frameworks

```
// image: [N,IC,IH,IW], filters: [OC,IC/GR,FH,FW]
// output: [N,OC,OH,OW], bias: [OC]
void conv_naive(float *image, float *filters, float *output,
                float *bias, int N, int IH, int IW, int IC,
                int FH, int FW, int OH, int OW, int OC, int SH,
                int SW, int DH, int DW, int GR) {
  int IC_GR = IC / GR, int OC_GR = OC / GR;

  for (int n = 0; n < N; ++n) {
    for (int gr = 0; gr < GR; ++gr) {
      for (int oc_gr = 0; oc_gr < OC_GR; ++oc_gr) {
        for (int oh = 0; oh < OH; ++oh) {
          for (int ow = 0; ow < OW; ++ow) {
            int oc = gr * OC_GR + oc_gr;
            float sum = bias != NULL ? bias[oc] : 0.0f;

            for (int ic_gr = 0; ic_gr < IC_GR; ++ic_gr) {
              for (int fh = 0; fh < filter_height; ++fh) {
                for (int fw = 0; fw < filter_width; ++fw) {
                  int ic = gr * IC_GR + ic_gr;
                  int ih = oh * SH + fh * DH;
                  int iw = ow * SW + fw * DW;
                  sum += image[n, ic, ih, iw] *
                         filters[oc, ic_gr, fh, fw];
                }
              }
            }
            output[n, oc, oh, ow] = sum;
          }
        }
      }
    }
  }
}
```

Listing 4.1: Naive convolution, assuming no padding for brevity. Tensors are accessed with Numpy-style indexing.

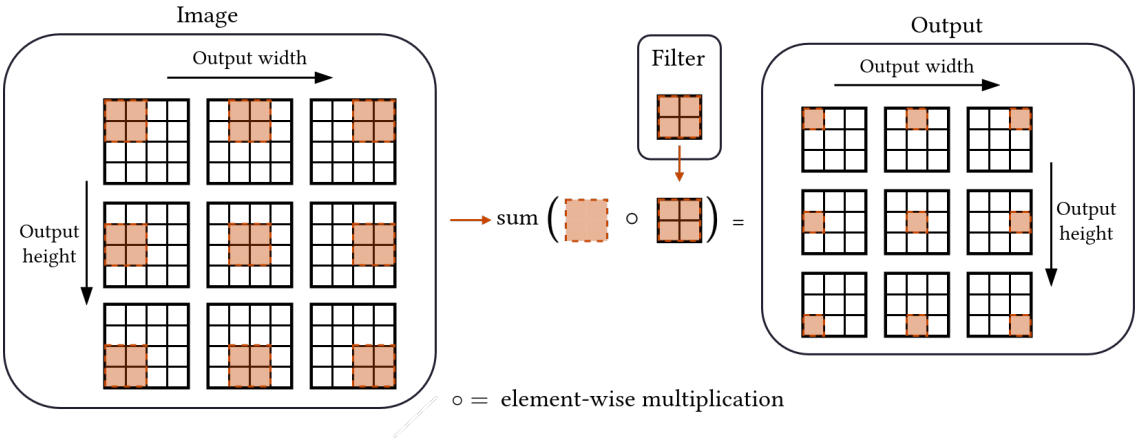
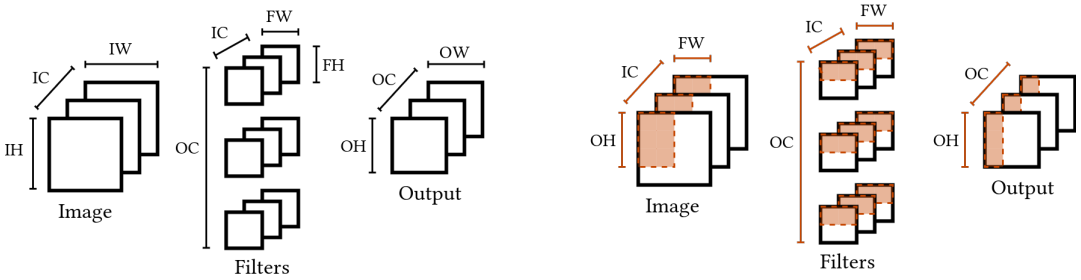


Figure 4.1: View of a naive convolution that uses unit stride, no padding, no dilation, and a 2×2 filter. The tensors have only height and width dimensions, the image is 4×4, and the output is 3×3. Black arrows indicate loop iterations, and colors indicate the data flow. The operation at the center is an element-wise multiplication followed by a sum reduction.



(a) Convolution input tensors

(b) ZConv GEMM dimensions example

Figure 4.2: On the left, the input tensors to the convolution and their respective dimensions (omitting the batch dimension). On the right, an example of a GEMM within ZConv, highlighting its input elements and dimensions.

such as PyTorch [3] and ONNX Runtime [23]. The Im2col transformation takes the patches of the input image (as highlighted in Figure 4.1) and unrolls them into columns (or rows) of a matrix, which is used as one of the GEMM inputs. When patches overlap, Im2col duplicates data, increasing the transformation overhead.

Despite their overhead, GEMM-based implementations remain widely used because GEMM is a core routine of the Level-3 BLAS [34], and has been heavily optimized in linear algebra libraries such as BLIS [81], OpenBLAS [82], and OneMKL [41]. On CPUs, these libraries approach peak performance through techniques such as tiling and loop reordering, data packing, and architecture-specific microkernels [33].

```
void cblas_sgemm (const CBLAS_LAYOUT layout,
                  const CBLAS_TRANSPOSE transa,
                  const CBLAS_TRANSPOSE transb,
                  const int m, const int n, const int k,
                  const float alpha, const float *a,
                  const int lda, const float *b, const int ldb,
                  const float beta, float *c, const int ldc);
```

Listing 4.2: The C BLAS API for float GEMM.

Additionally, with the popularization of AI workloads, GEMM benefits from dedicated hardware acceleration, now present in CPUs, GPUs, and specialized AI accelerators, further improving its efficiency [8], [17], [44]. GEMM is defined in (4.1), where α and β are scalar values, and $C_{m\times n}$, $A_{m\times k}$, and $B_{k\times n}$ are the matrix operands. Listing 4.2 shows the C BLAS Application Programming Interface (API) for performing GEMM with float elements. This API uses the following parameters: (i) *layout*: the memory layout of matrices (row-major or column-major), (ii) *transx*: whether to transpose matrix x before multiplication, (iii) m, n, k: the dimension sizes of the matrices, (iv) *alpha* and *beta*: the scalar multipliers parameters, (v) *ldx*: the leading dimension for matrix x, which for row-major matrices is the number of elements in memory between consecutive elements in a column, (vi) a, b, c: the base pointers to the matrices.

$$C_{m\times n} = \beta * C_{m\times n} + \alpha * A_{m\times k} \cdot B_{k\times n} \tag{4.1}$$

While Im2col transforms the convolution input image into a new contiguous matrix to leverage GEMM, the GEMM API (Listing 4.2) itself supports non-contiguous input matrices. In the row-major layout, the GEMM API only assumes that elements within a row are contiguous. The stride between rows, the row size, and the starting position of the matrix are all flexible and configurable through its parameters. The column-major layout operates similarly, but with contiguous columns instead of rows. BLAS libraries rely on tiling—dividing input matrices into smaller submatrices (tiles)—to optimize for cache locality [33]. These tiles are generally non-contiguous in memory and are made contiguous through data packing. Data packing copies these tiles into contiguous buffers and often reorders their elements, improving TLB locality and access patterns within the GEMM microkernel [76]. These transformations

are applied whether the original matrices are contiguous or not, thus, GEMM also achieves high performance with non-contiguous inputs. ZConv leverages GEMM's API flexibility by providing non-contiguous matrices directly to GEMM, eliminating the need for Im2col and achieving efficient convolution with minimal overhead.

4.2 Zero-Copy Convolution

Zero-Copy Convolution (ZConv) is a novel convolution algorithm for CPUs designed around the key insight that all data copies required by traditional Im2col-based approaches can be eliminated by decomposing the convolution into multiple GEMM calls and leveraging the GEMM API to operate on non-contiguous input matrices. As a result, ZConv performs convolutions with zero extra memory copies—aside from the internal packing transformations that may be performed by BLAS libraries. The only exceptions are grouped ($\texttt{GR} > 1$) and dilated ($\texttt{DH} > 1 \vee \texttt{DW} > 1$) convolutions, where a small copy is required. ZConv's performance relies on heavily optimized BLAS GEMM kernels, making its implementations simple and compatible with most architectures via open-source or vendor-specific BLAS libraries, such as OpenBLAS [82] and BLIS [81]. ZConv supports all convolution parameters used in PyTorch's *Conv2D* operator (listed in Table 4.1), except for the padding mode, which is fixed to zero-padding. For this, ZConv includes two versions: a zero-copy implementation that covers most common configurations (defined in Listing 4.3), and an extended version that introduces a small copy step to support grouped and dilated convolutions (defined in Listing 4.4). Line 2 of Listing 4.3 specifies the tensor layouts used by ZConv. These layouts are commonly found in deep learning frameworks such as PyTorch [3] and TensorFlow [64]. Sections 4.2.1 and 4.2.2 describe the algorithm and Section 4.2.3 describe its integration into PyTorch.

4.2.1 ZConv Algorithm

Listing 4.3 presents the ZConv algorithm. Although padding parameters are omitted and tensor indexing is simplified for readability, the listing closely matches its C language implementation. The two outermost loops iterate over the batch (`N`) and output width (`OH`), which correspond to the first and third dimensions of the output

tensor. Each iteration of these loops computes a distinct output tile of shape `[OH,1,OC]`, allowing these loops to be parallelized. In this case, they are parallelized via an OpenMP pragma directive in Line 7 [20]. This directive effectively merges the two outermost loops into a single loop and executes its iterations in parallel. Lines 11-13 initialize the output tile either to zero or to the corresponding bias values, if provided. The only remaining loop, at Line 15, iterates over the filter height (`FH`), setting the appropriate GEMM parameters and calling GEMM. Each iteration over the filter height computes a partial result for the `[OH,1,OC]` output tile. GEMM is called in row-major layout, with input matrices used as-is (*i.e.*, no transposition), and with scalar multipliers `alpha` and `beta` both set to one. The typical dimensions of the GEMM matrices are: `[OH,OC]` for `c`, `[OH,FW*IC]` for `a`, `[FW*IC,OC]` for `b`. Overall, GEMM is invoked $\texttt{N} * \texttt{OW} * \texttt{FH}$ times. An example of these matrix tiles within the convolution tensors is shown in Figure 4.2b. These tiles may be smaller because padding in ZConv is handled by skipping accesses to padding elements, resulting in GEMM calls with smaller matrices. Padding elements can be skipped and never instantiated in temporary buffers because zero-padding is assumed.

```
1  // image: [N,IH,IW,IC], filters: [FH,FW,IC,OC]
2  // output: [N,OH,OW,OC], bias: [OC]
3  void zconv(float *image, float *filters, float *output,
4             float *bias, int N, int IH, int IW, int IC, int FH,
5             int FW, int OH, int OW, int OC, int SH, int SW) {
6
7    #pragma omp parallel for collapse(2)
8    for (int n = 0; n < N; ++n) {
9      for (int ow = 0; ow < OW; ++ow) {
10
11       for (int oh = 0; oh < OH; ++oh)
12         for (int oc = 0; oc < OC; ++oc)
13           output[n, oh, ow, oc] = bias != NULL ? bias[oc] : 0.0f;
14
15       for (int fh = 0; fh < FH; ++fh) {
16         float *a = &image[n, fh, ow * SW, 0];
17         float *b = &filters[fh, 0, 0, 0];
18         float *c = &output[n, 0, ow, 0];
19         int dim_M = OH;
20         int dim_K = FW * IC;
21         int dim_N = OC;
22         int lda = IW * IC * SH;
23         int ldb = dim_N;
24         int ldc = OW * dim_N;
25         float alpha = 1.0f, beta = 1.0f;
26         cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
27                     dim_M, dim_N, dim_K, alpha, a, lda, b, ldb,
28                     beta, c, ldc);
29       }
30     }
31   }
32 }
```

Listing 4.3: ZConv algorithm, assumes no padding for brevity. Tensors are accessed with Numpy-style indexing.

Figure 4.3 illustrates the ZConv algorithm with a simplified example that uses the same convolution parameters as Figure 4.1. The figure expands the iterations over output width and filter height, corresponding to Lines 9 and 15 of Listing 4.3. The highlighted tiles in the image and filter represent the elements accessed during each iteration, which serve as inputs to the GEMM operation shown at the center of the figure. This GEMM accumulates its result into the corresponding tile of the output. No data copies are involved in selecting the tiles used in the GEMM —these tiles are determined using a combination of their base pointers, matrix dimensions, and leading dimensions, as shown in Lines 16-28 of Listing 4.3.

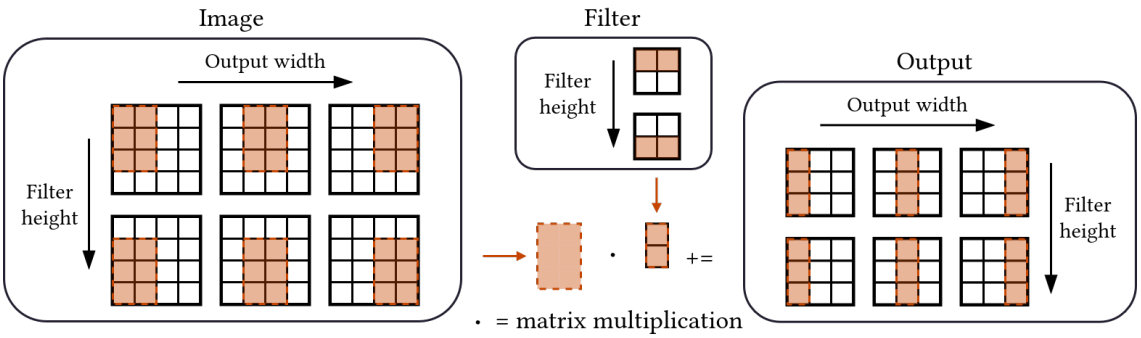


Figure 4.3: View of ZConv in a simplified convolution that uses unit stride, no padding, no dilation, and a 2x2 filter. The tensors have only height and width dimensions, the image is 4×4, and the output is 3×3. Black arrows indicate loop iterations, and colors indicate the data flow. The operation at the center is a GEMM, its output is accumulated into the output.

Intuitively, each GEMM call in ZConv is equivalent to sliding a single filter row vertically across the image while performing element-wise multiplication followed by a sum reduction between the filter row and the corresponding image elements. In Figure 4.3, the first filter row is never multiplied with the elements of the last image row, which is why the image tiles do not cover the full column height in this example. Similarly, the second filter row is not multiplied with the elements of the first image row. The horizontal iterations follow the naive convolution pattern but operate on taller image tiles. The algorithm builds upon the GEMMs shown this example, adding the input and output channels dimensions to its matrix operands. The GEMM call parameters allow directly using these contiguous and non-contiguous matrices present in the convolution tensors.

Lines 16-28 of Listing 4.3 define the GEMM call and its parameters. The base pointer for matrix `a` points to the beginning of an `[OH,FW*IC]` tile of the image tensor, following the iteration pattern illustrated in Figure 4.3. Matrix `a` is a non-contiguous tile, where each row contains `FW * C` contiguous elements, and the `lda` parameter specifies the stride between rows. Both the base pointer of `a` and its `lda` are affected by the stride parameters `SH` and `SW`. The algorithm naturally produces a matrix `c` of shape `[OH,OC]`. While this matrix could be stored as a contiguous tile in the output tensor, it would cause the output to have a transposed layout `[N,OW,OH,OC]`, where the `OH` and `OW` dimensions are swapped. Matrix `c` is stored as a non-contiguous

tile to avoid this transposed layout. In contrast, matrix `b` is a contiguous tile from the filter tensor. The filter tile shown in Figure 4.3 does not require transposition from row to column layout because a single row and a single column have the same representation in memory: a contiguous array.

4.2.2 ZConv Algorithm Support for Dilation and Grouped Convolution

Listing 4.4 presents the ZConv algorithm extended to support dilated and grouped convolution. As in the previous listing, padding is omitted and tensor indexing is simplified for readability. The two outermost loops, their parallelization, and the output tile initialization remain unchanged from the original algorithm. The following changes are introduced in the extended version: (i) An extra buffer of shape `[OH,FW∗IC/GR]` is allocated in Line 12; (ii) A new loop over the number of groups is added in Line 15; (iii) A copy operation from the image tensor to the extra buffer is added in Lines 16–25. The extra buffer is used as matrix `a` in the GEMM call.

```
// image: [N,IH,IW,IC], filters: [FH,FW,IC/GR,OC]
// output: [N,OH,OW,OC], bias: [OC]
void zconv_ext(float *image, float *filters, float *output,
               float *bias, int N, int IH, int IW, int IC, int FH,
               int FW, int OH, int OW, int OC, int SH, int SW,
               int DH, int DW, int GR) {
  int IC_GR = IC / GR, int OC_GR = OC / GR;

  #pragma omp parallel for collapse(2)
  for (int n = 0; n < N; ++n) {
    for (int ow = 0; ow < OW; ++ow) {
      float *a = (float *) malloc(OH * FW * IC_GR * sizeof(float));
      // The same [OH,1,OC] output tile initialization as ZConv
      for (int fh = 0; fh < FH; ++fh) {
        for (int gr = 0; gr < GR; ++gr) {
          int iw_start = ow * SW, iw_end = iw_start + FW * DW;
          int ih_start = fh * DH, ih_end = ih_start + OH * SH;
          int idx = 0;
          for (int h = ih_start; h < ih_end; h += SH) {
            for (int w = iw_start; w < iw_end; w += DW) {
              for (int ic_gr = 0; ic_gr < IC_GR; ++ic_gr) {
                a[idx++] = image[n, h, w, gr*IC_GR + ic_gr];
              }
            }
          }
          float *b = &filters[fh, 0, 0, gr*OC_GR];
          float *c = &output[n, 0, ow, gr*OC_GR];
          int dim_M = OH;
          int dim_K = FW * IC_GR;
          int dim_N = OC_GR;
          int lda = dim_K;
          int ldb = OC;
          int ldc = OW * OC;
          float alpha = 1.0f, beta = 1.0f;
          cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                      dim_M, dim_N, dim_K, alpha, a, lda, b, ldb,
                      beta, c, ldc);
        }
      }
      free(a);
    }
  }
}
```

Listing 4.4: Extended version of the ZConv algorithm to handle groups and dilation, assumes no padding for brevity. Colors highlight the main differences from ZConv. Tensors are accessed with Numpy-style indexing.

The extra buffer a is necessary to support dilated and grouped convolution. Recall that in a row-major GEMM, rows must be contiguous in memory, and in the original

algorithm, matrix a has a row size of FW $*$ IC. Dilation and grouping introduce irregular memory access patterns that break this contiguity. Dilation in the width dimension inserts offsets between the FW elements of a row, while grouping reduces the number of input channels accessed, such that only IC/GR elements are used per row instead of the full IC dimension. Even libraries like BLIS [81], which allow non-unit strides within rows, cannot support these cases because the resulting offsets are non-uniform, affecting the FW and IC components of a row differently. To resolve this, Lines 16-25 copy the required elements from the image tensor contiguously into the buffer a, considering stride, dilation, and grouping.

Unlike in the original algorithm, matrix b may point to a non-contiguous tile of the filter tensor in the grouped convolution case. This is because, when the number of groups is greater than one, each GEMM call processes only a subset of the output channels dimension (OC). Specifically, OC/GR per group, rather than the full output channel dimension. As a result, the base pointers to matrices b and c include an offset to the starting position of the current group, as shown in Lines 26 and 27, and the corresponding row offsets of these matrices are handled through the leading dimension parameters ldb and ldc.

4.2.3 Integration of ZConv with PyTorch

The integration of ZConv into PyTorch [3] enables its evaluation within end-to-end vision models (Section 4.3.3). PyTorch supports multiple convolution backends and includes a selector that chooses between them, generally based on convolution parameters. In this integration, ZConv is a new backend that is selected when the following conditions are met: (i) PyTorch is compiled with a BLAS library (to provide a GEMM implementation). (ii) The input image uses the channel-last layout [N,IH,IW,IC] [28], matching the layout expected by ZConv. (iii) The convolution is not grouped (GR $= 1$), not pointwise (FH $\neq 1 \vee$ FW $\neq 1$), and both output height and width dimensions are smaller than the number of input channels (OH $<$ IC $\wedge$ OW $<$ IC). The motivation for this heuristic is described in Section 4.3.2.

The filter tensor in the PyTorch channel-last layout [OC,FH,FW,IC] differs from the layout used by ZConv, [FH,FW,IC,OC]. To use the ZConv backend, the filter tensor must be transformed. This transformation can be performed either once

Table 4.2: Experimental configuration

CPU	i7-14700K
Cores Used	8 P-Cores
Frequency	3 GHz (locked)
Memory	64 GiB DDR5

Table 4.3: Software versions

PyTorch	2.5.1
OneDNN	3.5.3
OneMKL	2025.0.0
Blis	1.0

Table 4.4: Models and convolution layers per source

	TorchVision	Timm
Models	98	1149
Unique *Conv2D*	9826	

Table 4.5: Types of layers in the models of Table 4.4. Types are not mutually exclusive.

Conv2D Type	Layer Count
Pointwise	6072
Grouped	2457
Strided	1699

ahead-of-time when converting the model to the channel-last format or before each invocation of a ZConv convolution. Except for Section 4.3.3, the end-to-end model evaluation performs the layout transformation ahead of time (Section 4.3.3).

4.3 ZConv Evaluation

This performance evaluation of Zero-Copy Convolution demonstrates that ZConv outperforms previous approaches and achieves competitive results compared to state-of-the-art implementations, with performance gains depending on the convolution parameters. In end-to-end models, ZConv improves inference execution time by up to 1.8×. This evaluation includes the standalone operation (Section 4.3.2) and convolutions within end-to-end PyTorch models (Section 4.3.3).

4.3.1 Setup

Table 4.2 provides details on the machine configuration used for evaluation. The CPU has both performance and efficiency cores. Only the performance cores (P-Cores) are used. To minimize measurement variance, the CPU governor is set to *performance*, the frequency is locked at 3 GHz, and Intel's Hyper-Threading and Turbo Boost are disabled. The system runs Ubuntu 22.04.5 with Linux kernel 6.5.0. Table 4.3 lists the software versions used in the evaluation.

Two collections of vision models are used: *TorchVision* [62] and *Timm* [85], both

Table 4.6: Tensor layouts used by each method in Section 4.3.2.

	Image/Output Layout	Weight Layout
Im2col	`N,C,H,W`	`OC,IC,FH,FW`
Yaconv ZConv LibTorch-ZConv	`N,H,W,C`	`FH,FW,IC,OC`
LibTorch	`N,H,W,C`	`OC,FH,FW,IC`

of which provide models for PyTorch. Table 4.4 summarizes the number of unique models in each collection and the distinct convolution layers they contain. Models from TorchVision that do not take images as input (video and optical flow models) are not included in Table 4.4. Quantized models are omitted because they do not use PyTorch's standard *Conv2D* implementation. Table 4.5 categorizes the convolutions from Table 4.4 into relevant types for the following sections.

4.3.2 Standalone Convolutions

This experiment evaluates how ZConv compares to previous approaches and state-of-the-art implementations by measuring the execution time of standalone convolutional layers. The evaluated convolution methods are listed below, and Table 4.6 summarizes the input image, weight, and output layouts they use. For methods that rely on GEMM calls, including those in LibTorch, Intel's OneMKL is used as the BLAS library.

1. **Yaconv**, a convolution method introduced by Korostelev *et al.* [47] as an extension of the BLIS library. It supports only single-threaded execution and does not handle strided, dilated, or grouped convolutions. It uses the author's original implementation with modifications to prevent runtime errors and to remove extra generated elements that are not part of the output.
2. **Im2col**, a C implementation based on the Caffe framework that transforms convolution into GEMM operations through the Im2col data transformation [10], [43]. For pointwise *Conv2D* with unit stride and no padding, it skips the Im2col step. Parallelism is handled by the GEMM library.
3. **ZConv**, a C implementation of Zero-Copy Convolution that leverages OpenMP for parallelism. It also utilizes OneMKL's Just-In-Time (JIT) optimization for

small GEMM sizes, which is handled automatically by the library.

4. **LibTorch**, PyTorch's C++ API implementation of *Conv2D*, which is used when running models with convolution layers. It may call multiple convolution backends, including OneDNN. Unlike other methods that only perform convolution, LibTorch also allocates its output tensor.
5. **LibTorch-ZConv**, ZConv's implementation in PyTorch. It works similarly to LibTorch, but utilizing the new ZConv backend.

These methods were tested on all *Conv2D* layers listed in Table 4.4. Before measuring performance, their correctness was verified using LibTorch as the reference output. Only Yaconv produced incorrect results. Out of the 6369 *Conv2D* layers supported by Yaconv, 425 had incorrect results.

The experimental evaluation runs convolution layers and measures execution time using a custom benchmarking framework built with components from Google Benchmark [32] and ConvBench [1]. In this framework, all methods execute a layer in random order before moving to the next one. Each configuration is run at least ten times or until 0.5 seconds have elapsed, with a warm-up of 0.1 seconds. This process is repeated five or more times, and the mean execution time is recorded.

In the following comparison graphs, the x-axis bars represent the speedup of one method over another across different convolution layers. The legend indicates the number of speedup and slowdown cases, while the x-axis label specifies the total number of layers shown. The y-axis uses a logarithmic scale to accommodate the wide range of values. Unlike traditional speedup graphs, which constrain slowdown values between 1 and 0, both speedup and slowdown are unbounded and symmetrical. Bars are sorted by speedup. Speedup values are clipped at the 0.99 quantile, with the maximum value displayed at the top of the graph. Similarly, slowdown values are clipped at the 0.01 quantile, with the minimum value shown at the bottom of the graph.

The total number of layers shown on the x-axis may differ from the total in Table 4.4 due to omitted layers. The following conditions lead to the omission of a layer: (1) layers not supported by a method or where a method (Yaconv) produced incorrect results; (2) layers where the execution time difference is insignificant — the 95% confidence interval around the mean values that are being compared must not

Table 4.7: Parameters for convolution layers used in comparisons of memory hierarchy metrics

	N	IC	IH	IW	OC	FH	FW	PH	PW	SH	SW	DH	DW	GR
Layer 1	1	32	112	112	64	3	3	1	1	1	1	1	1	1
Layer 2	1	64	56	56	64	3	3	1	1	1	1	1	1	1
Layer 3	1	224	7	7	224	3	3	1	1	1	1	1	1	1
Layer 4	1	320	14	14	320	2	2	0	0	2	2	1	1	1
Layer 5	1	384	14	14	384	3	3	1	1	1	1	1	1	384
Layer 6	1	960	33	33	256	3	3	12	12	1	1	12	12	1

overlap for a comparison to be significant[1]; (3) layers filtered by a heuristic because executing them with the new method is not beneficial. The figure captions specify heuristic-based exclusions. A heuristic applied in all comparisons is the exclusion of pointwise layers. This is because pointwise convolutions can be computed directly as GEMM without additional transformations (*e.g.*, Im2col). ZConv is designed to extract multiple smaller GEMMs within convolutions that require the Im2col data layout transformation. Direct GEMM, when applicable, is a simpler and more efficient approach.

Furthermore, six representative convolution layers, described in Table 4.7, are used to illustrate the memory hierarchy effects of the compared methods. Layers 1 to 3 are more conventional convolution configurations with decreasing input height and width. Layer 4 is a strided convolution that has no overlap between image patches. Layer 5 is a grouped (depthwise) convolution, and Layer 6 represents a dilated convolution. Table 4.8 also shows the total number of times these layers appear in models from TorchVision and Timm. Although Layer 6 appears only once, similar configurations are present in the models evaluated in Section 4.3.3. For the tables with memory metrics in this section, results were collected using the Linux Perf tool [31]. The methodology for Perf runs was adjusted so that all compared methods execute the same number of iterations (with a minimum of one hundred iterations). Reported Perf events are the average of one hundred such runs.

Table 4.8: Occurrences of convolution layers used in comparisons of memory hierarchy metrics

	Occurrences
Layer 1	68
Layer 2	374
Layer 3	76
Layer 4	145
Layer 5	124
Layer 6	1

Table 4.9: Memory hierarchy metrics of **Yaconv / ZConv using BLIS** as BLAS

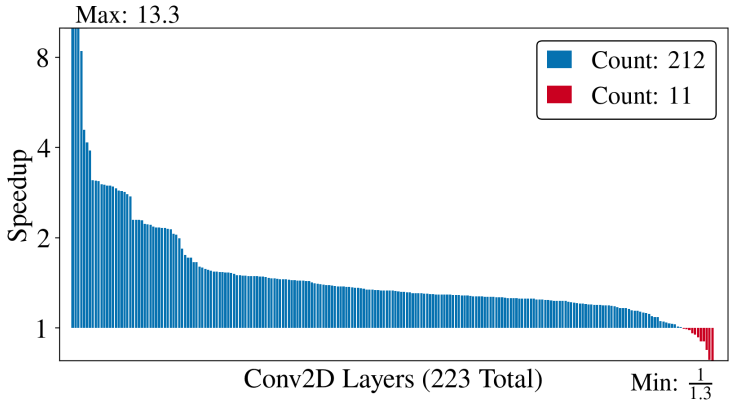
	Execution Time	Instructions	Load Misses				Page Faults
			L1	L2	L3	dTLB	
Layer 1	**1.3**	**1.3**	0.8	**8.4**	**16.6**	**77.5**	**278.0**
Layer 2	**1.3**	**1.3**	0.2	**17.2**	**26.4**	**70.4**	**345.6**

Yaconv

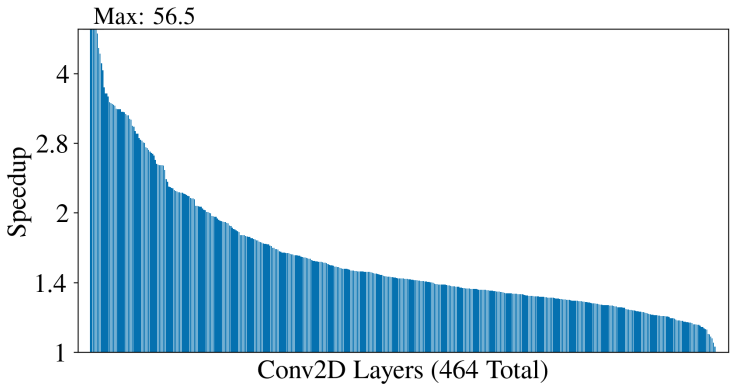
Figure 4.4 compares ZConv against Yaconv. For this comparison, ZConv is executed with a single thread because Yaconv does not support multithreading. While ZConv uses OneMKL as its BLAS library throughout the evaluation, Yaconv is integrated into the BLIS library. In Figure 4.4a, ZConv also utilizes BLIS as its BLAS provider for an accurate comparison of both methods. In Figure 4.4b, ZConv utilizes OneMKL. The results show significant improvements across almost all *Conv2D* layers, with a maximum speedup of 56×. Table 4.9 compares memory hierarchy metrics for the two layers from Table 4.7, which are supported by Yaconv. The values in Table 4.9 are relative to ZConv using BLIS, bolded values highlight metrics improved by ZConv.

Four main factors contribute to the performance gain and improved memory hierarchy metrics. (1) Because of its packing decisions, Yaconv performs additional computations that introduce extraneous elements in its output, requiring a temporary buffer to filter them out. In contrast, ZConv does not perform unnecessary computations and requires no copies in the *Conv2D* layers supported by Yaconv, except for packing that may be employed in the BLAS library. This factor is evidenced by ZConv executing fewer instructions, even though both algorithms execute similarly, and by also having better metrics in lower levels of memory in Table 4.9. (2) All

[1]For layers where the execution time difference is insignificant, the performance change is always less than 1%, meaning that no bars were excluded due to high variance in the measurements.



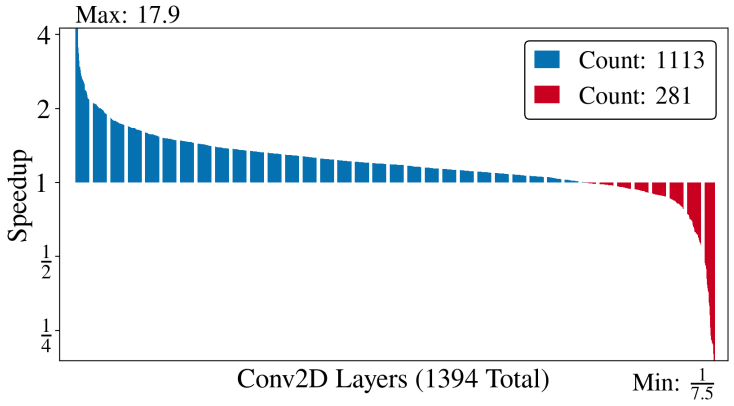
(a) ZConv using BLIS as BLAS



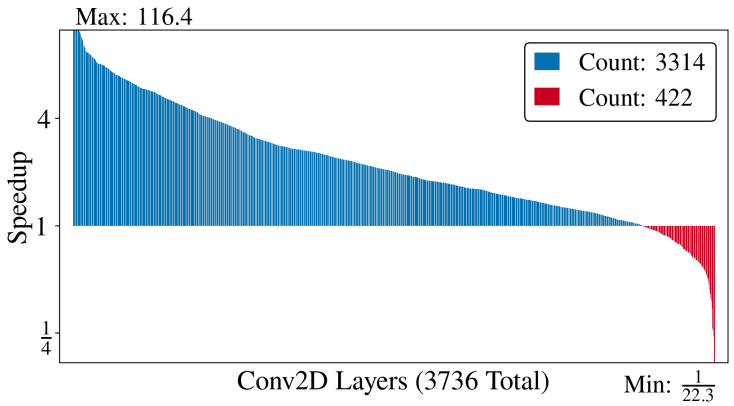
(b) ZConv using OneMKL as BLAS

Figure 4.4: Improvement of **ZConv over Yaconv** on individual *Conv2D* layers. Single-thread execution. The graphs exclude layers that are pointwise, unsupported by Yaconv, and for which Yaconv generated incorrect results. The graph showing ZConv using BLIS is included because Yaconv is integrated into BLIS.

optimizations (*e.g.*, tiling, packing, and loop ordering) in Yaconv, besides the GEMM microkernel, are manually implemented. In contrast, ZConv delegates most of these optimizations to the GEMM kernel, benefiting from its more advanced and flexible optimization strategies. (3) Yaconv's GEMM microkernel stores to the output in a strided fashion, with a stride of `OW` $*$ `M`. Even though ZConv utilizes non-contiguous tiles in the output tensor with the same stride, these tiles are given to the GEMM kernel (not the microkernel, as in Yaconv), which can better optimize for this access pattern. While Yaconv's tiling provides good L1 utilization, as shown in Table 4.9, the strided microkernel accesses likely cause the increased dTLB misses due to elements spanning multiple pages within a single tile. (4) Lastly, Yaconv is tightly integrated with the BLIS library, whereas ZConv is BLAS-agnostic. This flexibility



(a) Single-thread execution



(b) Multi-thread execution

Figure 4.5: Improvement of **ZConv over Im2col** on individual *Conv2D* layers. Both graphs exclude pointwise layers, and single-thread execution also excludes grouped layers.

allows ZConv to leverage vendor-optimized libraries such as OneMKL, as indicated by the higher speedups obtained in Figure 4.4b in comparison to Figure 4.4a.

Im2col

Figure 4.5 compares ZConv against Im2col for both single-threaded (Figure 4.5a) and multithreaded (Figure 4.5b) execution. The results show significant improvements with ZConv, particularly in multithreaded runs. Both graphs exclude pointwise layers because they are directly computed using GEMM, as previously discussed. Additionally, the single-threaded comparison excludes grouped convolution layers, which only benefit from ZConv when they are combined with ZConv's parallelization strategy. Table 4.10 shows the comparison of memory metrics between Im2col and

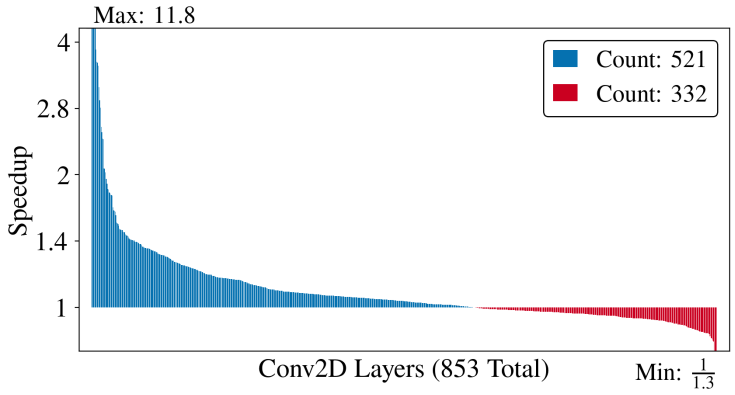
Table 4.10: Memory hierarchy metrics of **Im2col / ZConv**. Single-thread execution

	Execution Time	Instructions	Load Misses				Page Faults
			L1	L2	L3	dTLB	
Layer 1	**1.1**	**1.1**	**1.2**	**20.2**	**448.9**	**48456.0**	**13.7**
Layer 2	**1.1**	**1.3**	0.4	**15.1**	**93.3**	**16106.3**	**11.9**
Layer 3	**1.5**	**1.7**	0.9	0.3	**3.9**	**4.6**	**1.6**
Layer 4	1.0	**1.4**	0.2	0.1	0.5	**2.8**	**1.4**
Layer 5	0.2	0.3	0.4	**312.9**	**137.3**	**6.2**	**7.0**
Layer 6	**2.0**	**1.8**	0.3	0.2	**94.7**	0.8	**181.8**

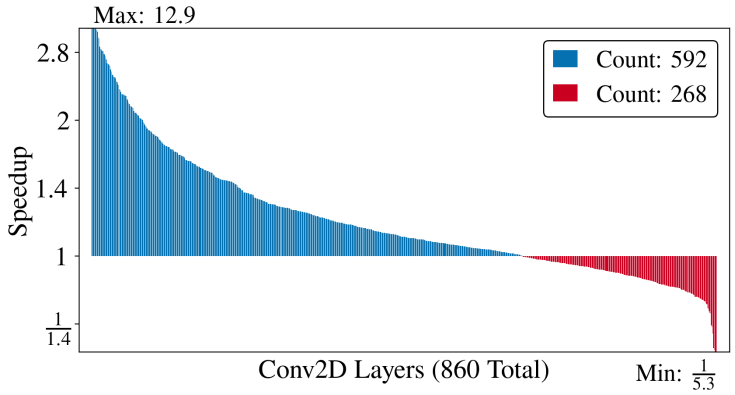
ZConv for single-threaded execution.

In grouped convolution, the channel dimension (`C`) is not accessed contiguously but rather in chunks of size `C`/groups. The most common case is depthwise convolution, where the number of groups equals the number of input channels. The extended version of ZConv used for grouped convolutions requires an additional copy operation to handle the blocked access pattern in the channel dimension. In contrast, Im2col benefits from using the `[N,C,H,W]` layout because the `C` dimension is non-contiguous in memory. However, the main disadvantage of ZConv in this scenario is that the input matrices for its GEMM calls become smaller, requiring more GEMM calls overall. Specifically, both the K and N GEMM dimensions are reduced by a factor of the number of groups, while the number of GEMM calls increases proportionally. Layer 5 in Table 4.10 is an example of depthwise convolution. In this case, the N GEMM dimension in ZConv is reduced to one, turning its GEMM calls into General Matrix-Vector Multiplication (GEMV). As a result, ZConv executes poorly vectorized code and incurs a significantly higher instruction count than Im2col. In particular, ZConv executes 138× more scalar instructions than Im2col (not shown in Table 4.10).

ZConv generally outperforms Im2col in multithreaded grouped convolution due to its effective parallelization strategy. Im2col relies on GEMM calls for parallelism, but its two outermost loops—over the batch dimension and the number of groups—run sequentially. The Im2col transformation is executed sequentially. In contrast, ZConv parallelizes its two outermost loops over the batch and the input width dimensions, ensuring that its entire computation runs in parallel, unlike Im2col. More advanced Im2col-based implementations, such as those used in LibTorch, improve parallelism



(a) Single-thread execution



(b) Multi-thread execution

Figure 4.6: Improvement of **LibTorch-ZConv over LibTorch** on individual *Conv2D* layers. Both graphs exclude pointwise grouped layers. Additionally, they exclude layers where output height or output width is greater than or equal to input channels.

by distributing the Im2col transformation across multiple threads.

The remaining slowdown cases in Figure 4.5, though less obvious than those previously discussed, can almost entirely be attributed to strided convolutions. In these cases, the Im2col transformation copies fewer or no redundant elements, reducing its overhead. Layer 4 in Table 4.10 is one such example, where Im2col has a reduced number of misses in all cache levels compared to ZConv. ZConv achieves better performance in the remaining cases by executing with no copies, redundant or otherwise, and still leveraging the highly optimized GEMM kernel, as evidenced by the improved memory hierarchy metrics for the remaining layers in Table 4.10.

Table 4.11: Memory hierarchy metrics of **LibTorch / LibTorch-ZConv**. Single-thread execution

	Execution Time	Instructions	Load Misses				Page Faults
			L1	L2	L3	dTLB	
Layer 1	0.9	0.9	0.3	**1.9**	0.4	0.7	0.9
Layer 2	1.0	0.9	0.4	**1.5**	0.4	0.7	**1.2**
Layer 3	**1.3**	**1.1**	**2.7**	**2.9**	0.3	0.2	1.0
Layer 4	**1.4**	1.0	**2.5**	**2.5**	**2.0**	**6.0**	1.0
Layer 5	0.1	0.1	0.3	**1.1**	**1.9**	0.5	0.9
Layer 6	**2.4**	**2.0**	**8.4**	**3.1**	**17.2**	**1.7**	**36.4**

LibTorch

Figure 4.6 compares LibTorch-ZConv against LibTorch for both single-threaded (Figure 4.6a) and multithreaded (Figure 4.6b) execution. These figures exclude pointwise and grouped convolutions, as well as layers where the output height or output width is greater than or equal to the number of input channels ($\texttt{OH} \geq \texttt{IC} \vee \texttt{OW} \geq \texttt{IC}$). Out of the 1404 total layers, excluding grouped and pointwise convolutions, 876 layers remain after applying this additional heuristic. Similar to the Im2col comparison, the results show overall good performance improvements, particularly in multithreaded runs. Table 4.11 shows the comparison of memory metrics between Libtorch and LibTorch-ZConv for single-threaded execution.

To understand the added heuristic (excluding layers where $\texttt{OH} \geq \texttt{IC} \vee \texttt{OW} \geq \texttt{IC}$), recall that GEMM is called $\texttt{FH} * \texttt{OW} * \texttt{N}$ times in ZConv, and the GEMM dimensions (assuming no padding) are: $M = \texttt{OH}$, $K = \texttt{FW} * \texttt{IC}$, $N = \texttt{OC}$. There are two concerns that the heuristic implicitly handles. First, it excludes layers where the K dimension would be too small compared to the other GEMM dimensions, as the IC is the main component of K (FW is rarely greater than a size of three). GEMM operations in linear algebra libraries are known to be less efficient on inputs with irregular sizes, such as having a small K dimension [91]. Second, the heuristic excludes layers that have an increased number of GEMM calls because this number increases proportionally to the OW size. Combining the two concerns, the filtered layers have an increased number of irregular GEMMs, leading to lower performance. A large majority of the filtered layers are the first few convolutional layers of the vision models, where the output height is large (*e.g.*, 112), but the number of input channels is small (*e.g.*, 3

in an Red Green Blue (RGB) image). One such example is Layer 1 in Table 4.11, where LibTorch has better performance and better metrics in all memory levels but L2 cache compared to LibTorch-ZConv. Another layer excluded is Layer 5, a grouped convolution. LibTorch has better performance in this layer for reasons previously discussed in Section 4.3.2.

LibTorch primarily utilizes its OneDNN backend, which used for all layers in Table 4.11, except Layer 3. OneDNN achieves great performance results from its highly optimized and target-specific templates designed for convolution, which are instantiated through JIT compilation based on input parameters and hardware characteristics [58]. OneDNN also leverages custom data layouts where the input tensors are reorganized for improved vectorization and efficient use of its batched GEMM microkernels [21], [38]. However, the OneDNN backend in LibTorch incurs costs seen in end-to-end model inference related to data layout transformations, JIT compilation, and loading of previously cached JIT-compiled code. These costs are amplified in multithread execution (Figure 4.6b) due to faster execution times, leading to higher speedups for LibTorch-ZConv. ZConv outperforms LibTorch's OneDNN backend in Layers 4 and 6 of Table 4.11. In Layer 6, which has large padding parameters, LibTorch-ZConv shows significant improvements over all memory metrics, largely due to its efficient padding handling, where padding elements (zeros) are never instantiated in temporary buffers. LibTorch also employs its native Im2col-based backend for some convolution configurations. Layer 3 in Table 4.11 is one example. As a result, the memory metrics for Layer 3 follow the same trends observed in the Im2col comparison.

4.3.3 End-to-end Models

This experiment evaluates the impact on end-to-end model inference time of integrating ZConv in PyTorch as an additional convolution backend. ZConv was integrated in C++ similarly to PyTorch's native Im2col-based convolution and leverages the GEMM library provided by PyTorch (OneMKL in this case). Unlike the standalone convolution evaluation in Section 4.3.2, which uses a C-based benchmarking framework, this section evaluates model inference using PyTorch's Python API. The Python API is preferred as it enables running models from TorchVision and Timm

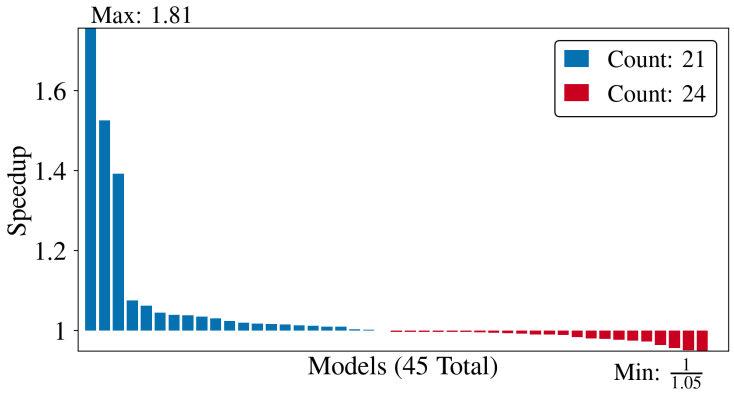
listed in Table 4.4. The models were evaluated under the following configurations:

1. **PyTorch**: PyTorch's default implementation. The model and input image are converted to the channel-last memory format before execution.
2. **PyTorch-ZConv**: The same as the previous configuration but with the following modifications. First, we added the new convolution backend implementing ZConv to PyTorch. Second, we modified the convolution backend selector in PyTorch to first check if it should use ZConv, if not, it proceeds with its default selection. The heuristic described in Section 4.3.2 is used for this selection. Lastly, when the model is converted to the channel-last format before execution, the weight layout of convolution layers using the ZConv backend is transformed to the `[FH,FW,IC,OC]` layout.

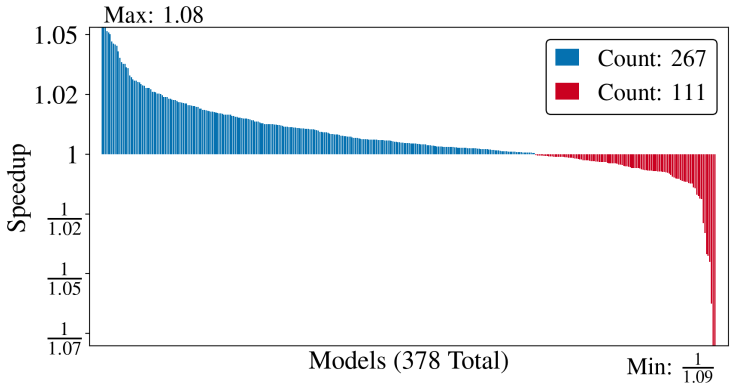
Each model is executed with both methods five times for warm-up, followed by fifty timed runs. The execution time of each non-warm-up run is recorded. This entire process is repeated at least five times to ensure consistent results. All models listed in Table 4.4 are run in inference mode using their respective default input sizes. For models from Timm, the default input size is obtained from the model's configuration. For TorchVision models, their respective preprocessing pipeline is applied to an input tensor of shape `[3,224,224]` to determine the default input size. All tests are executed with a batch size of 1.

The graphs in Figure 4.7 and Figure 4.8 follow a similar format to those in Section 4.3.2. However, in this case, the y-axis is in a linear scale, and the x-axis bars represent the speedup of the mean execution time of one method over another across different vision models, with the x-axis label indicating the total number of models shown. All models are included in the graphs except when the execution time difference between the two methods is insignificant. A comparison is considered significant only if the 95% confidence intervals around the means do not overlap. For models where the execution time difference is insignificant, the performance change is always less than 1%, meaning that no bars were excluded due to high variance in the measurements.

Figures 4.7 and 4.8 compare PyTorch-ZConv against PyTorch across Timm and TorchVision models for single-threaded and multithreaded execution, respectively. Notably, the speedup ranges vary between graphs, with wider ranges present in



(a) TorchVision models

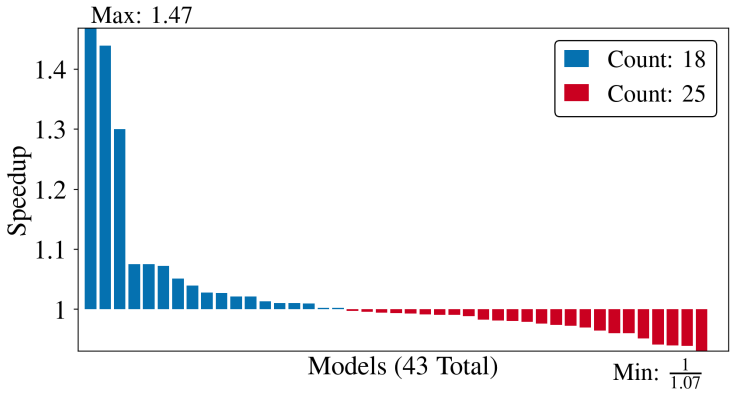


(b) Timm models

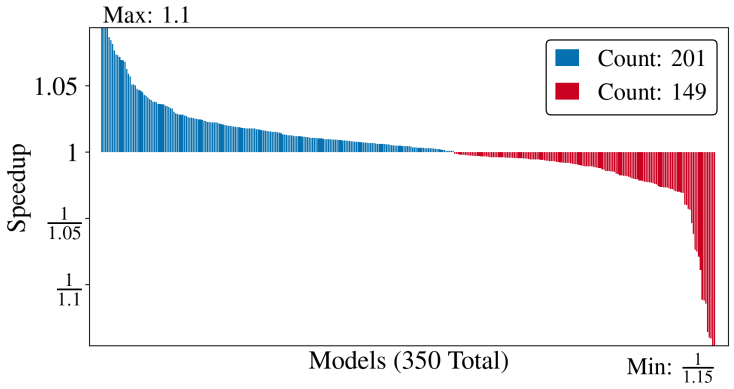
Figure 4.7: Improvement of **PyTorch-ZConv over PyTorch** on end-to-end models with single-thread execution.

the TorchVision results. Overall, the speedup trends are less pronounced than in Section 4.3.2 due to additional operations present in these models, such as pooling, and the fact that not all convolutional layers utilize ZConv. Across all results, the highest observed speedup is 1.81×, while the largest slowdown is 1.15 ×. The impact of ZConv on performance depends on the specific combination of convolutional layers present in each model.

The first three models in the TorchVision graphs stand out for obtaining the highest speedups with ZConv. These bars correspond to the following models in both single-threaded and multithreaded graphs: *deeplabv3_mobilenet_v3_large*, *deeplabv3_resnet50*, and *deeplabv3_resnet101*. DeepLabV3 is a segmentation model architecture that utilizes dilated convolution and supports distinct backbones, such as *resnet* and *mobilenet* [12]. Dilated convolutional layers typically have longer



(a) TorchVision models



(b) Timm models

Figure 4.8: Improvement of **PyTorch-ZConv over PyTorch** on end-to-end models with multi-thread execution.

execution times due to their poor memory access patterns, as the elements within an input patch are non-contiguous in memory. In DeepLabV3 models, dilated layers also have large padding sizes. ZConv's approach is to gather the necessary input elements into a contiguous buffer before executing a GEMM call. While Im2col has a similar approach, ZConv minimizes the number of non-contiguous accesses because it only considers the width dilation in the copy to the new buffer. The height dilation only affects the loop that iterates over the filter height (`FH`). Combined with its efficient padding handling, as discussed in Section 4.3.2, ZConv significantly improves performance for these layers.

Table 4.12: Total execution time of running inference on all images in a validation set of the Pascal VOC 2012 dataset with *DeepLabV3+* models using PyTorch and PyTorch-ZConv.

DeepLabV3+ Backbone	PyTorch Time (s)	PyTorch-ZConv Time (s)	Speedup
Xception	213	189	**1.13**
Mobilenet	92	89	**1.03**
Resnet50	269	216	**1.25**
Resnet101	341	289	**1.18**
hrnetv2_32	589	456	**1.29**
hrnetv2_48	883	678	**1.30**

DeepLabV3+ Models

Given that *DeepLabV3* models had the highest speedups with ZConv, Table 4.12 extends the evaluation to the more recent *DeepLabV3+* models [13]. While Timm and TorchVision do not provide these models, their PyTorch implementations are available on GitHub [27]. Unlike the previous end-to-end model evaluations, these models are evaluated following the usage of the repository that implements them. Thus, each model is run on a validation set of the Pascal VOC 2012 dataset, which contains 1449 images [26]. Table 4.12 presents details on the evaluated *DeepLabV3+* model backbones, the total inference time for multithreaded execution across all validation images, and the corresponding speedup obtained by PyTorch-ZConv. For a simpler integration, the weight layouts remained unchanged after converting the model to the channel-last format, requiring ZConv to convert the weight tensor to (FH, FW, IC, OC) before execution. Despite this additional transformation overhead, ZConv improves execution time across all models, further demonstrating its effectiveness for this architecture.

4.4 Discussion

GEMM-based convolution remains a popular approach due to the widespread availability and heavy optimization of BLAS libraries, despite the computational and memory overhead of the commonly used Im2col transformation. This chapter introduces the ZConv algorithm, which removes the need for Im2col by enabling GEMM calls to access convolution inputs directly. ZConv improves upon prior

GEMM-based methods by requiring no data copies, relying on unmodified GEMM kernels, and supporting a wide range of convolution configurations. As demonstrated in the evaluation, both ZConv and its extended version achieve significant speedups over prior work and state-of-the-art PyTorch implementations. The evaluation also defines simple heuristics based on the convolution parameters to filter convolution configurations that do not benefit from executing with ZConv. When integrated into PyTorch, ZConv achieves up to 1.8× speedup in end-to-end model inference, being particularly effective in image segmentation models that utilize dilated convolution. Future work includes adapting the algorithm to target GPU execution.

Chapter 5

Related Work

The work presented in this thesis investigates different aspects of data layout transformations. This chapter is organized by different research areas related to the projects presented in Chapters 2, 3, and 4.

5.1 Data Packing Optimizations

Earlier work proposed copying, a simpler version of packing that does not change data layout. Lam *et al.* investigate a copying approach to reduce cache interference misses and demonstrate its use for matrix multiplication [49]. Nevertheless, as discussed by Lam *et al.*, and later by Temam *et al.* [79], copying overhead is not always compensated for by performance improvements. Temam *et al.* pioneered a compile-time technique to determine *what* and *when* to apply copying [79]. In the same year, Esseghir proposed a Tile-and-Copy algorithm that couples tiling and copying [25]. However, these previous works only consider copying to reduce cache interference. To the best of our knowledge, GPAT is the first general compiler analysis and code transformation that targets TLB utilization through packing.

Coleman *et al.* present a tile-size selection (TSS) algorithm that decreases cache interference misses while avoiding copying [18]. Their work acknowledges that TLB misses have a higher performance impact than cache interference misses. But TSS avoids TLB misses by restricting the size and shape of tiles, which is not possible for some computations, for example the fused loop nest of the *2mm* benchmark (Section 2.3.3). GPAT overcomes this challenge by changing the packed tensor's data layout to improve access order. GPAT also detects cases where most accesses are

already contiguous and where packing would have a small, or even negative, effect, such as for some tile sizes of *gemm* (Section 2.3.3).

On modern CPUs, cache interference became less significant when compared to TLB misses due to higher cache associativity and larger problem sizes [78]. Park *et al.* acknowledge the importance of TLB misses and present a mathematical model to estimate TLB utilization in blocked algorithms [70]. TLB misses are reduced with a novel storage layout, *blocked data layout* (BDL), which partitions and reorders sub-blocks of the entire input matrix to ensure memory contiguity. Different from GPAT, the approach of BDL is not designed as a compiler transformation. Instead, BDL is a language-level construct that requires a matrix to be explicitly constructed in BDL prior to any computation. This approach restricts computations to a single tiling strategy because the block size must match the BDL. In contrast, GPAT finds opportunities and applies packing to n-dimensional tensors regardless of tile size and data layout. GPAT also allows a tensor to be packed at multiple points in a computation, having a different data-layout change at each point. Others recognize the impact of data-layout transformations on the performance of tensor computations [45], [46]. Nevertheless, they only consider data-layout changes for the entire tensor and thus suffer from TLB-miss penalties for large tensors.

Packing is mostly used in high-performance BLAS libraries [36], [81], [88]. A generic packing interface is exposed by these libraries but needs to be explicitly used when implementing new operations. Such a framework limits the use of packing to these library-supported operations and requires highly specialized developers. Such limitations are also recognized by Li *et al.* [57]. In particular, Li *et al.* show evidence that the approach adopted by all BLAS libraries is not adequate for tall-and-skinny matrices that arise from convolution computations [11]. In contrast, GPAT is a general compiler analysis and transformation that makes packing transparently available to all programmers — and computations apart from those in BLAS — thus eliminating the requirement that one should know how to use packing within specialized libraries.

Uday Bondhugula is a pioneer in attempting to match the performance of GEMM libraries with compiler-generated code in MLIR [5]. Bondhugula's study starts with a naïve hand-written implementation of GEMM in MLIR Affine dialect. The

level-3 loop nest is then tiled and reordered using an external polyhedral library. For packing, Bondhugula's approach utilizes the *data copy generation* (DCG) pass in Affine, which GPAT builds upon. However, the current implementation of DCG lacks a cost-benefit analysis such as the one integrated in GPAT. As a result, to prevent packing all tensors, Bondhugula hand-tuned the DCG pass for GEMM. In contrast, GPAT is fully agnostic to the computation in a target loop nest. Other code generation approaches either do not implement packing as a compiler pass or only consider packing for GEMM [65], [84].

5.2 Data Structure Splicing

Mannarwammy's RBSL [63] has significantly influenced this chapter and is closely related to RebaseDL. Similar to RBSL, RebaseDL distinguishes itself from the broader literature on structure layout transformation by adopting region-based transformations rather than whole-program transformations. The major limitation of RBSL is the absence of a reuse analysis. Specifically, both RBSL and RebaseDL calculate the total benefit of their respective transformed layouts by multiplying the benefit with the number of iterations of an 'ancestor loop', depicted in Equation 3.7 in this chapter. However, their work does not require accesses to the transformed data to be invariant to the ancestor loop, only the existence of the ancestor loop is required. This lack of invariance, and consequently data reuse, can lead to a situation in which the cost of the copy is never amortized by the improved data layout. We contacted the authors but, unfortunately, their implementation is not available and their performance improvements remain challenging to reproduce.

The IA-64 (Itanium) processor used in Mannarwammy *et al.*'s evaluation has a Very Long Instruction Word (VLIW) Instruction Set Architecture (ISA). In such architectures, the compiler's ability to generate an efficient schedule is key to performant code. Thus, we speculate that a significant percentage of the improvements reported in Mannarwammy *et al.*'s work, especially for regions with little or no reuse, stem from the fact that transformed loops exhibit simpler memory access patterns that the compiler can exploit to generate better schedules. RebaseDL solves this problem by incorporating a reuse analysis in the first phase of its analysis. Furthermore, the

definition of RebaseDL's transformation differs from RBSL's because of the data reuse requirement. Beyond these primary distinctions, RebaseDL computes cache utilization per candidate, whereas RBSL also computes it per type, and RebaseDL considers control-flow divergence in Phase 4, RBSL does not.

Curial *et al.* describe an implementation of whole-program maximal structure splitting in the IBM XL compiler [19]. They use Pool Allocation [51] to transform arbitrarily allocated link-based data structures into pool regions and apply structure-splitting transformations to these pools. An earlier work by Zhao *et al.* describes maximal splitting for arrays of structures [95]. Neither of these approaches describes a cost analysis similar to RebaseDL that takes into consideration data reuse. Also, in both approaches, the imprecision of the available pointer analysis is a major limitation to the applicability of the transformation, providing further motivation for the region-based approach described in this chapter.

Chapter 2 proposed a compile-time analysis and transformation that performs data packing, focusing on data-layout changes and TLB utilization [76]. In contrast, RebaseDL utilizes cache utilization for its cost model, and it applies to a broader class of programs where loop tiling and multi-dimensional arrays may not be present. Furthermore, the data packing by RebaseDL can be applied to strided accesses to an array, while data packing in these previous works is applied mostly to tensor tiles.

5.3 GEMM-Based Convolutional Layers

Chellapilla *et al.* [10] introduced the popular Im2col method for computing convolution with GEMM. As explained in Section 4.1, this method transforms the image tensor via data copying and data duplication to harness high-performance GEMM implementations. Popularized by the Caffe framework [43], this method is still employed by modern deep learning frameworks such as PyTorch [3] and Onnx Runtime [23] with modifications to lower the overhead of Im2col. PyTorch parallelizes the transformation, and Onnx Runtime may fuse Im2col with convolution to then tile and parallelize the computation. However, these state-of-the-art frameworks do not rely only on Im2col-based convolution, they use hybrid algorithms that change the convolution implementations based on input parameters and other constraints such

as target hardware and tensor layouts. ZConv's approach leverages high-performance GEMM kernels and eliminates the need for data layout transformations in most convolution configurations. By doing so, the evaluation in Section 4.3 shows that ZConv can outperform both the traditional Im2col approach and state-of-the-art implementations in PyTorch, and that it can speedup end-to-end models when integrated into PyTorch's hybrid algorithm.

MEC [16], SMM-Conv [69], and Yaconv [47] reduce convolution to GEMM similarly to ZConv. MEC reduces vertical redundancies present in Im2col, whereas SMM-Conv and Yaconv share the same insight that implicit smaller matrix multiplications are present in convolution when isolating a filter row (or column), and these can be executed as GEMM operations. MEC reduces a convolution to either $\texttt{OH}$ or $\texttt{N} * \texttt{OH}$ matrix multiplications, and SMM-Conv to $\texttt{N} * \texttt{IC} * \texttt{OW} * \texttt{FH}$ matrix multiplications. Yaconv reduces a convolution into several calls to a GEMM microkernel. ZConv addresses several limitations found in these prior works: (i) MEC requires a large extra buffer of size $\texttt{N} * \texttt{OW} * \texttt{IH} * \texttt{FW} * \texttt{IC}$, executing similarly to Im2col with a less redundant extra buffer. SMM-Conv allocates a packing buffer of size $\texttt{IH} * \texttt{OW}$ per thread for its image tensor. Yaconv's output includes extraneous elements. Removing these elements requires an extra buffer of size $\texttt{OH} * \texttt{OW} * \texttt{OC} + Ex$ and a copy operation, where Ex represents the number of extraneous elements. In the same convolution configurations supported by these approaches, ZConv requires no additional buffers beyond the packing that may be employed by high-performance BLAS libraries [33]. (ii) Yaconv and SMM-Conv lack support for dilated, grouped, and strided ($\texttt{SH} > 1 \vee \texttt{SW} > 1$) convolutions, and MEC lacks support for dilated and grouped convolutions. ZConv supports all these convolution variants. (iii) Yaconv's algorithm defines its loop ordering, tiling, and packing optimizations, except for the GEMM microkernel, which is provided by the BLIS library. This approach makes its algorithm complex and tightly coupled to the BLIS library. SMM-Conv, on the other hand, does not utilize a GEMM kernel likely due to the small size of the matrices it multiplies, lacking target-specific optimizations such as a GEMM microkernel. ZConv is a more portable approach, relying entirely on GEMM kernels for optimizations and being compatible with any BLAS library (*e.g.*, OpenBLAS [82], BLIS [81]). This separation also allows ZConv to achieve better performance, as shown in Section 4.3.2.

(iv) Yaconv does not support parallel execution, and SMM-Conv and MEC may utilize separate parallel steps in their execution, whereas ZConv is fully fused into a single loop nest and trivially parallelized. (v) Lastly, only ZConv is evaluated in a large set of convolution layers, which allows designing a heuristic to integrate it into a hybrid convolution approach, such as that used by PyTorch (Section 4.3.2).

Another approach used in prior work is to fuse the Im2col transformation into the GEMM operation itself [24], [77], [83], [96]. Dukhan [24] introduced the Indirect Convolution, which modifies the GEMM routine used in an Im2col implementation to access the image tensor directly via an indirection buffer. This technique avoids the explicit Im2col transformation at the cost of additional overhead in the form of extra pointer indirections within the GEMM microkernel. San Juan *et al.* [77] integrate Im2col into the packing step of GEMM, generating packed tiles directly from the image tensor. Their implementation utilizes the BLIS library [81] with modifications to its matrix B packing logic to perform this implicit Im2col. While these methods reduce the overhead associated with Im2col, they require modifications to the GEMM implementation. In contrast, ZConv eliminates the Im2col transformation and relies on unmodified GEMM kernels that access the image tensor directly.

Anderson *et al.* [2] explored a wide range of Im2col variants and introduced two novel GEMM-based convolution algorithms that do not rely on the Im2col transformation. Their key insight is that a conventional convolution, which uses a filter tensor with dimensions FH and FW, can be decomposed into $\mathsf{FH} * \mathsf{FW}$ pointwise convolutions. As discussed earlier, pointwise convolution can be directly implemented as a GEMM operation. However, in this formulation, the image tensor must be shifted before each GEMM call, depending on which filter tile is being applied. The two algorithms proposed in their work require additional memory of size $\mathsf{OC} * \mathsf{OH} * \mathsf{OW}$ and $\mathsf{FH} * \mathsf{OW} + \mathsf{IC} * \mathsf{IH} + \mathsf{IC} * \mathsf{IW}$, respectively, and both depend on the parallelism offered by the GEMM kernels. The latter algorithm reduces memory overhead by setting the GEMM kernel to accumulate each partial result directly into the final output tensor. However, this requires allocating an output tensor with extra space and performing a "hole-punching" step between GEMM calls, in which certain elements of the image tensor are temporarily zeroed—an operation that falls outside the parallel GEMM execution. ZConv requires no auxiliary memory and fully parallelizes its entire

computation. Moreover, while Anderson *et al.* [2] do not address how their methods can support strided, padded, dilated, or grouped convolutions, ZConv is designed to handle all of these convolution variants.

Georganas *et al.* [30] introduced a novel batch-reduce GEMM kernel and demonstrated its suitability for implementing deep learning (DL) primitives. Intel's state-of-the-art OneDNN framework combines the batch-reduce microkernel with JIT compilation, among other optimization techniques [58]. As seen in Section 4.3.2, this strategy achieves great performance in many convolution configurations. However, OneDNN's approach has limited compatibility with the data layouts commonly used in DL frameworks such as PyTorch, requiring additional data transformations. Consequently, PyTorch is unable to fully extract the performance offered by OneDNN. ZConv relies on standard GEMM kernels, making it a more portable solution due to the widespread availability of high-performance GEMM implementations. Its reliance on conventional GEMM also simplifies integration into existing DL frameworks.

Chapter 6

Conclusion

The cache hierarchy used in modern CPUs is designed around the principles of temporal and spatial locality. These properties enable caches to operate transparently to programmers. Region-based data layout transformations simulate a software-controlled cache within this hardware-controlled hierarchy by inserting explicit data copies into specific code regions and then changing memory accesses to the copied data. The effectiveness of these transformations depends on data reuse, which helps to overcome the overhead introduced by copying. This thesis demonstrates that when applied through careful compiler analysis (Chapters 2 and 3), these transformations can drastically reduce code execution time. At runtime, similar strategies are a core component of high-performance GEMM implementations. They improve data locality even for non-contiguous matrix inputs, leveraging the inherent data reuse in GEMM computations. Composing such GEMM operations with non-contiguous inputs enabled the design of a new high-performance convolution algorithm (Chapter 4). This new convolution algorithm demonstrates that region-based data layout transformations are effective in the development of high-performance algorithms.

Chapter 2 presents GPAT, a compiler analysis and transformation for packing. The analysis determines what to pack, where to insert packing within a loop nest, and whether packing yields a performance benefit. Experiments show that GPAT unlocks many optimization opportunities in the PolyBench/C benchmarks, achieving speedups of up to 4× over modern loop optimizers such as Polly and Pluto. Furthermore, GPAT can complement these optimizers when used in combination. Its implementation in MLIR allows widespread use of the technique. Recent frameworks,

such as ALT [89], illustrate growing interest in data layout transformations. Building on the separated computation and schedule paradigm found in Halide [74] and TVM [14], ALT introduces scheduling directives that allow joint optimization of loop structure and data layout. It employs search and cost models to find high-performance schedules. However, such frameworks lack a static analysis to guide their transformations, such as the one presented in GPAT. The ideas behind GPAT can enhance traditional loop optimizers (*e.g.*, Polly, Pluto), DL frameworks (*e.g.*, TVM, ALT), and general-purpose compilers.

Chapter 3 presents RebaseDL, a compiler analysis for region-based structure splitting and field reordering. This analysis identifies code regions that present data reuse and suffer from poor cache utilization due to large memory strides in their accesses. In the SPEC CPU benchmark suite, RebaseDL identifies many such regions and achieves speedups of up to 1.34× in them. Implemented in LLVM, RebaseDL applies to general-purpose code, where loop bounds may not be statically known and memory accesses often involve aliasing. These challenges limit GPAT's applicability, but the analysis remains valuable: it highlights profitable transformation opportunities and can be extended with runtime guards or manual inspection to mitigate dynamic conditions.

Lastly, Chapter 4 introduces ZConv, a GEMM-based convolution algorithm that fully eliminates the need for the Im2col transformation and other temporary copies. ZConv relies on unmodified GEMM kernels and exploits their internal data packing to achieve high performance. The results demonstrate that ZConv outperforms both prior Im2col-based methods and state-of-the-art implementations in PyTorch. It achieves up to 12.9× speedup on isolated convolution benchmarks against PyTorch implementations, and up to 1.8× improvement in end-to-end inference when integrated into PyTorch. In addition to its performance, ZConv is highly portable and simple to implement, enabling easy integration into existing frameworks.

References

[1] L. Alvarenga, V. Ferrari, R. Souza, M. Pereira, and G. Araujo, *ConvBench: A comprehensive benchmark for 2D convolution primitive evaluation*, 2024. arXiv: 2407.10730. [Online]. Available: https://arxiv.org/abs/2407.10730.

[2] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "High-performance low-memory lowering: GEMM-based algorithms for DNN convolution," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 99–106. DOI: 10.1109/SBAC-PAD49847.2020.00024. [Online]. Available: https://doi.org/10.1109/SBAC-PAD49847.2020.00024.

[3] J. Ansel, E. Yang, H. He, *et al.*, "PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, La Jolla, CA, USA: Association for Computing Machinery (ACM), 2024, pp. 929–947, ISBN: 9798400703850. DOI: 10.1145/3620665.3640366. [Online]. Available: https://doi.org/10.1145/3620665.3640366.

[4] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of recurrences–a method to expedite the evaluation of closed-form functions," in *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, Oxford, United Kingdom, 1994, pp. 242–249. DOI: 10.1145/190347.190423. [Online]. Available: https://doi.org/10.1145/190347.190423.

[5] U. Bondhugula, *High performance code generation in MLIR: An early case study with GEMM*, 2020. arXiv: 2003.00532. [Online]. Available: http://arxiv.org/abs/2003.00532.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *International Conference on Programming Language Design and Implementation (PLDI)*, Tucson, AZ, USA: Association for Computing Machinery (ACM), 2008, pp. 101–113. [Online]. Available: https://doi.org/10.1145/1379022.1375595.

[7] R. Bruno, V. Jovanovic, C. Wimmer, and G. Alonso, "Compiler-assisted object inlining with value fields," in *International Conference on Programming Language Design and Implementation (PLDI)*, Virtual, Canada, 2021, pp. 128–141. DOI: 10.1145/3453483.3454034. [Online]. Available: https://doi.org/10.1145/3453483.3454034.

[8] J. P. L. de Carvalho, J. E. Moreira, and J. N. Amaral, "Compiling for the IBM matrix engine for enterprise workloads," *IEEE Micro*, vol. 42, no. 5, pp. 34–40, 2022. DOI: 10.1109/MM.2022.3176529. [Online]. Available: https://doi.org/10.1109/MM.2022.3176529.

[9] A. Castelló, J. Bellavita, G. Dinh, Y. Ikarashi, and H. Martínez, "Tackling the matrix multiplication micro-kernel generation with exo," in *CGO*, Edinburgh, United Kingdom: IEEE Press, 2024, pp. 182–192. DOI: 10.1109/CGO57630.2024.10444883. [Online]. Available: https://doi.org/10.1109/CGO57630.2024.10444883.

[10] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France): Suvisoft, 2006. [Online]. Available: https://inria.hal.science/inria-00112631.

[11] J. Chen, N. Xiong, X. Liang, *et al.*, "TSM2: Optimizing tall-and-skinny matrix-matrix multiplication on GPUs," in *International Conference on Supercomputing (ICS)*, Phoenix, Arizona, USA: Association for Computing Machinery (ACM), 2019, pp. 106–116. [Online]. Available: https://doi.org/10.1145/3330345.3330355.

[12] L. Chen, G. Papandreou, F. Schroff, and H. Adam, *Rethinking atrous convolution for semantic image segmentation*, 2017. arXiv: 1706.05587. [Online]. Available: http://arxiv.org/abs/1706.05587.

[13] L. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, *Encoder-decoder with atrous separable convolution for semantic image segmentation*, 2018. arXiv: 1802.02611. [Online]. Available: http://arxiv.org/abs/1802.02611.

[14] T. Chen, T. Moreau, Z. Jiang, *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594. [Online]. Available: https://doi.org/10.48550/arXiv.1802.04799.

[15] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in *International Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, USA, 1999, pp. 13–24. DOI: 10.1145/301618.301635. [Online]. Available: https://doi.org/10.1145/301618.301635.

[16] M. Cho and D. Brand, "MEC: Memory-efficient convolution for deep neural network," in *International Conference on Machine Learning (ICML*, Sydney, NSW, Australia: JMLR.org, 2017, pp. 815–824. [Online]. Available: https://doi.org/10.5555/3305381.3305466.

[17] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 tensor core GPU: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, 2021. DOI: 10.1109/MM.2021.3061394. [Online]. Available: https://doi.org/10.1109/MM.2021.3061394.

[18] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," *ACM SIGPLAN Notices*, vol. 30, no. 6, pp. 279–290, Jun. 1995. [Online]. Available: https://doi.org/10.1145/207110.207162.

[19] S. Curial, P. Zhao, J. N. Amaral, *et al.*, "MPADS: Memory-pooling-assisted data splitting," in *International Symposium on Memory Management (ISMM)*, Tucson, AZ, USA, 2008, pp. 101–110. DOI: 10.1145/1375634.1375649. [Online]. Available: https://doi.org/10.1145/1375634.1375649.

[20] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998. DOI: 10.1109/99.660313. [Online]. Available: https://doi.org/10.1109/99.660313.

[21] D. Das, S. Avancha, D. Mudigere, *et al.*, *Distributed deep learning using synchronous stochastic gradient descent*, 2016. arXiv: 1602.06709. [Online]. Available: http://arxiv.org/abs/1602.06709.

[22] J. P. L. De Carvalho, B. Kuzma, I. Korostelev, *et al.*, "KernelFaRer: Replacing native-code idioms with high-performance library calls," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 3, pp. 1–22, Sep. 2021. [Online]. Available: https://doi.org/10.1145/3459010.

[23] O. R. developers, *ONNX runtime*, Version: 1.21.0, 2021. [Online]. Available: https://onnxruntime.ai/.

[24] M. Dukhan, *The indirect convolution algorithm*, 2019. arXiv: 1907.02129. [Online]. Available: http://arxiv.org/abs/1907.02129.

[25] K. Esseghir, "Improving data locality for caches," M.S. thesis, Rice University, 1993.

[26] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (VOC) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, Jun. 2010. [Online]. Available: https://doi.org/10.1007/s11263-009-0275-4.

[27] G. Fang, *DeepLabV3Plus-Pytorch*, 2020. [Online]. Available: https://github.com/VainF/DeepLabV3Plus-Pytorch.

[28] V. Fedyunin, *Channels last memory format in PyTorch*, 2020. [Online]. Available: https://pytorch.org/tutorials/intermediate/memory%5C%5Fformat%5C%5Ftutorial.html.

[29] V. Ferrari, R. Sousa, M. Pereira, *et al.*, "Advancing direct convolution using convolution slicing optimization and ISA extensions," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 4, Dec. 2023, ISSN: 1544-3566. DOI: 10.1145/3625004. [Online]. Available: https://doi.org/10.1145/3625004.

[30] E. Georganas, K. Banerjee, D. Kalamkar, *et al.*, "Harnessing deep learning via a single building block," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 222–233. DOI: 10.1109/IPDPS47924.2020.00032. [Online]. Available: https://doi.org/10.1109/IPDPS47924.2020.00032.

[31] T. Gleixner and I. Molnar, *Linux perf*. [Online]. Available: https://perfwiki.github.io/main/.

[32] Google, *Benchmark*, 2016. [Online]. Available: https://github.com/google/benchmark.

[33] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Software*, vol. 34, no. 3, May 2008, ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. [Online]. Available: https://doi.org/10.1145/1356052.1356053.

[34] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 BLAS," *ACM Trans. Math. Software*, vol. 35, no. 1, Jul. 2008, ISSN: 0098-3500. DOI: 10.1145/1377603.1377607. [Online]. Available: https://doi.org/10.1145/1377603.1377607.

[35] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly - performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 4, Dec. 2012. [Online]. Available: https://doi.org/10.1142/S0129626412500107.

[36] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, 2010. [Online]. Available: http://eigen.tuxfamily.org.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90.

[38] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "LIBXSMM: Accelerating small matrix multiplications by runtime code generation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Salt Lake City, Utah: IEEE Computer Society Press, 2016, ISBN: 9781467388153. DOI: 10.5555/3014904.3015017. [Online]. Available: https://doi.org/10.5555/3014904.3015017.

[39] A. Howard, M. Sandler, B. Chen, *et al.*, "Searching for MobileNetV3," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2019, pp. 1314–1324. DOI: 10.1109/ICCV.2019.00140. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCV.2019.00140.

[40] R. Hundt, S. Mannarswamy, and D. Chakrabarti, "Practical structure layout optimization and advice," in *International Symposium on Code Generation and Optimization (CGO)*, New York, New York, USA, 2006, pp. 233–244. DOI: 10.1109/CGO.2006.29. [Online]. Available: https://doi.org/10.1109/CGO.2006.29.

[41] Intel, *oneAPI math kernel library (OneMKL)*, 1994. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[42] B. Jähne, *Digital Image Processing*. Germany: Springer Berlin Heidelberg, 2005, ISBN: 9783540240358. DOI: 10.1007/3-540-27563-0. [Online]. Available: https://doi.org/10.1007/3-540-27563-0.

[43] Y. Jia, E. Shelhamer, J. Donahue, *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *International Conference on Multimedia (MM)*, Orlando, Florida, USA: Association for Computing Machinery (ACM), 2014, pp. 675–678, ISBN: 9781450330633. DOI: 10.1145/2647868.2654889. [Online]. Available: https://doi.org/10.1145/2647868.2654889.

[44] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture*, Toronto, ON, Canada: Association for Computing Machinery (ACM), 2017, pp. 1–12, ISBN: 9781450348928. DOI: 10.1145/3079856.3080246. [Online]. Available: https://doi.org/10.1145/3079856.3080246.

[45] I. Kadayif and M. Kandemir, "Quasidynamic layout optimizations for improving data locality," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 15, no. 11, pp. 996–1011, Nov. 2004. [Online]. Available: https://doi.org/10.1109/TPDS.2004.70.

[46] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "Improving locality using loop and data transformations in an integrated framework," in *International Symposium on Microarchitecture (MICRO)*, Dallas, Texas, USA: IEEE Computer Society Press, 1998, pp. 285–296. [Online]. Available: https://doi.org/10.1109/MICRO.1998.742790.

[47] I. Korostelev, J. P. L. De Carvalho, J. Moreira, and J. N. Amaral, "YaConv: Convolution with low cache footprint," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 1, Feb. 2023, ISSN: 1544-3566. DOI: 10.1145/3570305. [Online]. Available: https://doi.org/10.1145/3570305.

[48] A. Kumar and S. Pop, "SCoP detection: A fast algorithm for industrial compilers," in *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Prague, Czech Republic: Association for Computing Machinery (ACM), 2016. [Online]. Available: https://acohen.gitlabpages.inria.fr/impact/impact2016/papers/impact2016-kumar.pdf.

[49] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *ACM SIGPLAN Notices*, vol. 26, no. 4, pp. 63–74, Apr. 1991. [Online]. Available: https://doi.org/10.1145/106973.106981.

[50] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, California, USA: IEEE Computer Society Press, 2004, pp. 75–86. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665.

[51] C. Lattner and V. Adve, "Automatic pool allocation: Improving performance by controlling data structure layout in the heap," in *International Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, USA, 2005, pp. 129–142. DOI: 10.1145/1065010.1065027. [Online]. Available: https://doi.org/10.1145/1065010.1065027.

[52] C. Lattner, M. Amini, U. Bondhugula, *et al.*, "MLIR: Scaling compiler infrastructure for domain specific computation," in *International Symposium on Code Generation and Optimization (CGO)*, Seoul, Republic of Korea: IEEE Computer Society Press, 2021, pp. 2–14. [Online]. Available: https://doi.org/10.1109/CGO51591.2021.9370308.

[53] R. Lavaee, "The hardness of data packing," in *Symposium on Principles of Programming Languages (POPL)*, St. Petersburg, FL, USA, 2016, pp. 232–242. DOI: 10.1145/2837614.2837669. [Online]. Available: https://doi.org/10.1145/2837614.2837669.

[54] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4013–4021. DOI: 10.1109/CVPR.2016.435. [Online]. Available: https://doi.org/10.1109/CVPR.2016.435.

[55] Y. LeCun, B. Boser, J. S. Denker, *et al.*, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989. DOI: 10.1162/neco.1989.1.4.541. [Online]. Available: https://doi.org/10.1162/neco.1989.1.4.541.

[56] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791. [Online]. Available: https://doi.org/10.1109/5.726791.

[57] C. Li, H. Jia, H. Cao, *et al.*, "AutoTSMM: An auto-tuning framework for building high-performance tall-and-skinny matrix-matrix multiplication on CPUs," in *International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, New York City, NY, USA: IEEE Computer Society Press,

2021, pp. 159–166. [Online]. Available: https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00034.

[58] J. Li, Z. Qin, Y. Mei, *et al.*, "Onednn graph compiler: A hybrid approach for high-performance deep learning compilation," in *International Symposium on Code Generation and Optimization (CGO)*, Edinburgh, United Kingdom: IEEE Press, 2024, pp. 460–470, ISBN: 9798350395099. DOI: 10.1109/CGO57630.2024.10444871. [Online]. Available: https://doi.org/10.1109/CGO57630.2024.10444871.

[59] X. Liu, K. Sharma, and J. Mellor-Crummey, "ArrayTool: A lightweight profiler to guide array regrouping," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Edmonton, AB, Canada, 2014, pp. 405–416. DOI: 10.1145/2628071.2628102. [Online]. Available: https://doi.org/10.1145/2628071.2628102.

[60] *LLVM block frequency terminology*. [Online]. Available: https://llvm.org/docs/BlockFrequencyTerminology.html.

[61] *LLVM loop terminology*. [Online]. Available: https://llvm.org/docs/LoopTerminology.html.

[62] T. maintainers and contributors, *TorchVision: PyTorch's computer vision library*, 2016. [Online]. Available: https://github.com/pytorch/vision.

[63] S. S. Mannarswamy, R. Govindarajan, and R. Surendran, "Region based structure layout optimization by selective data copying," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Raleigh, North Carolina, USA, 2009, pp. 338–347. DOI: 10.1109/PACT.2009.43. [Online]. Available: https://10.1109/PACT.2009.43.

[64] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. [Online]. Available: https://www.tensorflow.org/.

[65] J. Meng, C. Zhuang, P. Chen, *et al.*, "Automatic generation of high-performance convolution kernels on ARM CPUs for deep learning," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 11, pp. 2885–2899, Nov. 2022. [Online]. Available: https://doi.org/10.1109/TPDS.2022.3146257.

[66] N. Mitchell, L. Carter, J. Ferrante, and K. Hogstedt, "Quantifying the multi-level nature of tiling interactions," *International Journal of Parallel Programming*, vol. 26, no. 6, pp. 641–670, Dec. 1998. [Online]. Available: https://doi.org/10.1023/A:1018782528453.

[67] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising C to polyhedral MLIR," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Atlanta, GA, USA: IEEE Computer Society Press, 2021, pp. 45–59. [Online]. Available: https://doi.org/10.1109/PACT52795.2021.00011.

[68] S. Muchnick, *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[69] A. Ofir and G. Ben-Artzi, "SMM-Conv: Scalar matrix multiplication with zero packing for accelerated convolution," in *Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Los Alamitos, CA, USA: IEEE Computer Society Press, Jun. 2022, pp. 3066–3074. DOI: 10.1109/CVPRW56347.2022.00346. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPRW56347.2022.00346.

[70] N. Park, B. Hong, and V. Prasanna, "Tiling, block data layout, and memory hierarchy performance," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 14, no. 7, pp. 640–654, Jul. 2003. [Online]. Available: https://doi.org/10.1109/TPDS.2003.1214317.

[71] E. Petrank and D. Rawitz, "The hardness of cache conscious data placement," in *Symposium on Principles of Programming Languages (POPL)*, Portland, Oregon, 2002, pp. 101–112. DOI: 10.1145/503272.503283. [Online]. Available: https://doi.org/10.1145/503272.503283.

[72] S. Pop, A. Cohen, and G.-A. Silber, "Induction variable analysis with delayed abstractions," in *International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, Barcelona, Spain, 2005, pp. 218–232. DOI: 10.1007/11587514_15. [Online]. Available: https://doi.org/10.1007/11587514%5C%5F15.

[73] L.-N. Pouchet and T. Yuki, *Polybench: The polyhedral benchmark suite (version 4.2.1)*, 2016. [Online]. Available: https://sourceforge.net/projects/polybench/.

[74] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *International Conference on Programming Language Design and Implementation (PLDI)*, Seattle, Washington, USA: Association for Computing Machinery (ACM), 2013, pp. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176.

[75] C. S. Rohwedder, J. P. L. de Carvalho, and J. N. Amaral, "Artifact of "region-based data layout via data reuse analysis"," Zenodo, Jan. 2024. DOI: 10.5281/zenodo.10457086. [Online]. Available: https://doi.org/10.5281/zenodo.10457086.

[76] C. Salvador Rohwedder, N. Henderson, J. P. L. De Carvalho, Y. Chen, and J. N. Amaral, "To pack or not to pack: A generalized packing analysis and transformation," in *International Symposium on Code Generation and Optimization (CGO)*, Montréal, QC, Canada: Association for Computing Machinery (ACM), 2023, pp. 14–27, ISBN: 9798400701016. DOI: 10.1145/3579990.3580024. [Online]. Available: https://doi.org/10.1145/3579990.3580024.

[77] P. San Juan, A. Castelló, M. F. Dolz, P. Alonso-Jordá, and E. S. Quintana-Ortí, "High performance and portable convolution operators for multicore processors," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 91–98. DOI: 10.1109/SBAC-PAD49847.2020.00023. [Online]. Available: https://doi.org/10.1109/SBAC-PAD49847.2020.00023.

[78] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based TLB preloading," in *ACM SIGARCH Computer Architecture News*, Vancouver, British Columbia, Canada: Association for Computing Machinery (ACM), 2000, pp. 117–127. [Online]. Available: https://doi.org/10.1145/342001.339666.

[79] O. Temam, E. D. Granston, and W. Jalby, "To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts," in *Conference on Supercomputing*, Portland, Oregon, USA: Association for Computing Machinery (ACM), 1993, pp. 410–419. [Online]. Available: https://doi.org/10.1145/169627.169762.

[80] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing*, Berlin, Heidelberg, 2010, pp. 157–173. DOI: 10.1007/978-3-642-11261-4_11. [Online]. Available: https://doi.org/10.1007/978-3-642-11261-4%5C%5F11.

[81] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Software*, vol. 41, no. 3, Jun. 2015, ISSN: 0098-3500. DOI: 10.1145/2764454. [Online]. Available: https://doi.org/10.1145/2764454.

[82] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs," in *International Conference on High Performance Computing, Networking, Storage and Analysis*, Denver, Colorado: Association for Computing Machinery (ACM), 2013, ISBN: 9781450323789. DOI: 10.1145/2503210.2503219. [Online]. Available: https://doi.org/10.1145/2503210.2503219.

[83] Q. Wang, S. Mei, J. Liu, and C. Gong, "Parallel convolution algorithm using implicit matrix multiplication on multi-core CPUs," in *International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–7. DOI: 10.1109/IJCNN.2019.8852012. [Online]. Available: https://doi.org/10.1109/IJCNN.2019.8852012.

[84] B. Wasti, J. P. Cambronero, B. Steiner, H. Leather, and A. Zlateski, *LoopStack: A lightweight tensor algebra compiler stack*, 2020. arXiv: 2205.00618. [Online]. Available: http://arxiv.org/abs/2205.00618.

[85] R. Wightman, *Timm: PyTorch image models*, 2019. [Online]. Available: https://github.com/huggingface/pytorch-image-models.

[86] C. Wimmer and H. Mössenböck, "Automatic array inlining in java virtual machines," in *International Symposium on Code Generation and Optimization (CGO)*, Boston, MA, USA, 2008, pp. 14–23. DOI: 10.1145/1356058.1356061. [Online]. Available: https://doi.org/10.1145/1356058.1356061.

[87] X. Wu, M. Kruse, P. Balaprakash, *et al.*, "Autotuning polybench benchmarks with LLVM clang/polly loop optimization pragmas using bayesian optimization," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, GA, USA: IEEE Computer Society Press, 2020, pp. 61–70. [Online]. Available: https://doi.org/10.1109/PMBS51919.2020.00012.

[88] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 BLAS performance optimization on loongson 3A processor," in *International Conference on Parallel and Distributed Systems (ICPADS)*, Singapore: IEEE Computer Society Press, 2012, pp. 684–691. [Online]. Available: https://doi.org/10.1109/ICPADS.2012.97.

[89] Z. Xu, J. Xu, H. Peng, *et al.*, "ALT: Breaking the wall between data layout and loop optimizations for deep learning compilation," in *European Conference on Computer Systems*, Rome, Italy: Association for Computing Machinery, 2023, pp. 199–214, ISBN: 9781450394871. DOI: 10.1145/3552326.3587440. [Online]. Available: https://doi.org/10.1145/3552326.3587440.

[90] J. Yan, J. He, W. Chen, P.-C. Yew, and W. Zheng, "ASLOP: A field-access affinity-based structure data layout optimizer," *Science China Information Sciences*, vol. 54, no. 9, pp. 1769–1783, 2011. DOI: 10.1007/s11432-011-4265-0. [Online]. Available: https://doi.org/10.1007/s11432-011-4265-0.

[91] W. Yang, J. Fang, D. Dong, X. Su, and Z. Wang, "LIBSHALOM: Optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, St. Louis, Missouri: Association for Computing Machinery (ACM), 2021, ISBN: 9781450384421. DOI: 10.1145/3458817.3476217. [Online]. Available: https://doi.org/10.1145/3458817.3476217.

[92] L. Ye, M. Lis, and A. Fedorova, "A unifying abstraction for data structure splicing," in *International Symposium on Memory Systems (MEMSYS)*, Washington, District of Columbia, USA, 2019, pp. 173–183. DOI: 10.1145/3357526.3357548. [Online]. Available: https://doi.org/10.1145/3357526.3357548.

[93] C. Yu, P. Roy, Y. Bai, H. Yang, and X. Liu, "LWPTool: A lightweight profiler to guide data layout optimization," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, no. 11, pp. 2489–2502, 2018. DOI: 10.1109/TPDS.2018.2840992. [Online]. Available: https://doi.org/10.1109/TPDS.2018.2840992.

[94] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *International Conference on Machine Learning*, J. Dy and A. Krause, Eds., vol. 80, PMLR, Jul. 2018, pp. 5776–5785. [Online]. Available: https://proceedings.mlr.press/v80/zhang18d.html.

[95] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral, "Forma: A framework for safe automatic array reshaping," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 1, 2–es, Nov. 2007. DOI: 10.1145/1290520.1290522. [Online]. Available: https://doi.org/10.1145/1290520.1290522.

[96] T. Zhao, Q. Hu, X. He, *et al.*, "ECBC: Efficient convolution via blocked columnizing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 1, pp. 433–445, 2023. DOI: 10.1109/TNNLS.2021.3095276. [Online]. Available: https://doi.org/10.1109/TNNLS.2021.3095276.

[97] L. Zheng, C. Jia, M. Sun, *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *Conference on Operating Systems Design and Implementation (OSDI)*, USA: USENIX Association, 2020, ISBN: 978-1-939133-19-9. [Online]. Available: https://doi.org/10.5555/3488766.3488815.

[98] Y. Zhong, M. Orlovich, X. Shen, and C. Ding, "Array regrouping and structure splitting using whole-program reference affinity," in *International Conference on Programming Language Design and Implementation (PLDI)*, Washington DC, USA, 2004, pp. 255–266. DOI: 10.1145/996841.996872. [Online]. Available: https://doi.org/10.1145/996841.996872.

[99] A. Zlateski, Z. Jia, K. Li, and F. Durand, "The anatomy of efficient FFT and winograd convolutions on modern CPUs," in *International Conference on Supercomputing (ICS)*, Phoenix, Arizona: Association for Computing Machinery (ACM), 2019, pp. 414–424, ISBN: 9781450360791. DOI: 10.1145/3330345.3330382. [Online]. Available: https://doi.org/10.1145/3330345.3330382.