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

Caio Salvador Rohwedder
csalvado@ualberta.ca
University of Alberta
Edmonton, Canada

Nathan Henderson
João P. L. De Carvalho
{nthender,labegali}@ualberta.ca
University of Alberta
Edmonton, Canada

Yufei Chen
José Nelson Amaral
{yufei24,jamaral}@ualberta.ca
University of Alberta
Edmonton, Canada

Abstract

Packing is an essential loop optimization for handcrafting a high-performance General Matrix Multiplication (GEMM). Packing copies a non-contiguous block of data to a contiguous block to reduce the number of TLB entries required to access it, avoiding expensive TLB misses. When copying data, packing can rearrange elements of the block to decrease the stride between consecutive accesses, improving spatial locality. Until now the use of packing has been limited to handcrafted GEMM implementations and to auto-tuning techniques. Existing loop optimizers, such as Polly and Pluto, either only apply packing to GEMM computations (Polly), or not at all (Pluto). This work proposes GPAT, a generalized packing analysis and code transformation that applies packing, when beneficial, to a generic input loop nest. GPAT is implemented in the Affine dialect of MLIR and evaluated on Polybench/C. GPAT applies packing to benchmarks beyond GEMM and obtains significant speedup compared to current loop optimizers that do not apply packing.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: data-copying, TLB, data-layout transformation

ACM Reference Format:
Caio Salvador Rohwedder, Nathan Henderson, João P. L. De Carvalho, Yufei Chen, and José Nelson Amaral. 2023. To Pack or Not to Pack: A Generalized Packing Analysis and Transformation. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579990.3580024

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0101-6/23/02…$15.00
https://doi.org/10.1145/3579990.3580024

1 Introduction

In large part, the success of numerical computations is due to the development of very efficient linear-algebra libraries [12, 29, 32]. A sophisticated implementation of a **General Matrix Multiplication** (GEMM) kernel that relies upon two transformations – tiling and packing – is key to libraries' efficiency [9, 10]. 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 tiling 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 [4], Halide [25], the Affine dialect of MLIR [18], and Polyhedral frameworks (*e.g.* Pluto [2] and Polly [11]), and optimizing compilers in general. TVM and the MLIR Affine dialect support explicit memory transfers designed for software-controlled memory buffers found in 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 [1]. 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 [6]. Wu *et al.* [31] 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 paper. 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 work 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.* [9] to optimize GEMM and GPAT's code transformation builds upon the infrastructure created by Bondhugula [1] to experiment with packing in GEMM.

The main contributions of this work 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 3)
- An open-source artifact that implements GPAT in the MLIR Affine dialect (Section 3.5)
- The evaluation of GPAT. It shows that GPAT's heuristic selects good combinations of packings, achieving significant speedup over the Polly [11] and Pluto [2] loop optimizers in the Polybench [24] 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 4).

2 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[1] occur when accessing the packed subtensor because contiguous elements are less likely mapped to the same cache set [16]. 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 [28]. 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.

Goto *et al.* [9] 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 [12], OpenBLAS [32], and BLIS [29] 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 [1, 30, 31], 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 remarked by Lam *et al.* [16] and Temam *et al.* [28], the copying overhead of an unrestricted packing algorithm that packs all tensors can easily outweigh its gains.

3 Packing Optimizing Analysis

The main contribution of this work 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.

[1] Commonly referred to as *conflict misses*.

3.1 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 control-flow graph.

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.

3.2 Analysis Constraints

Consistent with the practice in widely adopted IRs, GPAT's analysis requires `for` loops to be **Static Control Parts** (SCoPs) [15] 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.

```
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]
```

Listing 1. 3D Tensor Contraction: $C_{i,j} = \sum_k \sum_l A_{k,l,i} \times B_{l,k,j}$. On the right, the shape of the working set of A is shown outside of the loop nest and inside each loop.

```
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   for(j=0; j<60; j++)             // A'_{1x80x100}
8    for(k=0; k<80; k++)            // A'_{1x1x100}
9     for(l=0; l<100; l++)          // A'_{1x1x1}
10     a = load A'[0][k][l]
11     b = load B[l][k][j]
12     prod = mul a, b
13     c = load C[i][j]
14     sum = add c, prod
15     store sum, C[i][j]
```

Listing 2. Packing applied to 3D Tensor Contraction. On the right, the working set of `A'` is shown.

3.3 Running Example

The 3-dimensional tensor contraction represented in a pseudo-IR code in Listing 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 shows a packing transformation applied to Listing 1. Loops are referred to by the name of their IV, *e.g.* $\mathtt{for}_J$ is the loop in Line 3 of Listing 1. Listing 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 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 1, similarly, the working-set shape of `A'` is shown in Listing 2.

3.4 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 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 1 has four candidates (L, T) that exhibit reuse: $\{(\texttt{for}_J, \texttt{A}), (\texttt{for}_I, \texttt{B}), (\texttt{for}_K, \texttt{C}), (\texttt{for}_L, \texttt{C})\}$. In the $(\texttt{for}_J, \texttt{A})$ candidate, which is packed in Listing 2, all elements of `A'` are reused in every iteration of $\texttt{for}_J$ because the load to `A` in Line 6 of Listing 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 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. [21].

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 4.3.1).

Assume Listing 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 is discussed in Section 3.3.

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 empirically-driven 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 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 1, the data-layout change of `A'` ensures that accesses have unit stride across consecutive iterations of $\texttt{for}_L$ in Listing 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 3 describes a Boolean function to determine goal fulfillment.

```
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 3. Function that checks if a packing candidate achieves the goal of reducing TLB misses.

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 3 estimates the 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 1, `A` has a working set of shape $\texttt{A}_{1\times100\times1}$ in $\texttt{for}_K$, whereas, in Listing 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 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 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 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.

3.5 GPAT in the MLIR Affine Dialect

MLIR [18] is a recent addition to the LLVM compiler infrastructure project [17]. 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.

4 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 [2] 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 [11] 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?

4.1 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[2] benchmark suite [24]. 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 work employs Polygeist[3], a research C/C++ frontend that generates MLIR SCF & Affine Dialect code [22] 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.

4.2 Experimental Methodology

The results presented in Section 4.3 used the Google Benchmark[4] tool [8] to collect the average execution time of 100 executions of each evaluated approach. Section 4.3 employs the above methodology to evaluate: (i) **Polymer**[5]: an MLIR tool that extracts SCoPs of a program and optimizes the loops within each SCoP using Pluto [22], 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**[6]: packing applied by GPAT, which takes Polymer-optimized IR as input and (iii) **Individual Packings**: packing applied individually to 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 4.4 employs a similar methodology but uses the built-in Polybench/C timing facilities. The following approaches are evaluated in Section 4.4: (i) **Polly**[7]: an LLVM IR polyhedral loop optimizer [11]; (ii) **Clang -O3**: Clang[7] compilation with its optimization level 3 (-O3); (iii) **Polymer**;

[2] Version 4.2.1. [3] Commit 6ba6b7b8ac07c9d60994eb46b46682a9f76ea34e.
[4] Version 1.6.1. [5] Commit 4bb0aa2ff43f70bb3f13f221ebcf9f76fb41fa76.
[6] Commit packing-v0.5 [7] Version 14.0.1

(iv) **Affine**: the MLIR Affine dialect tiling pass and (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 3.4.

4.3 Packing Selection Evaluation

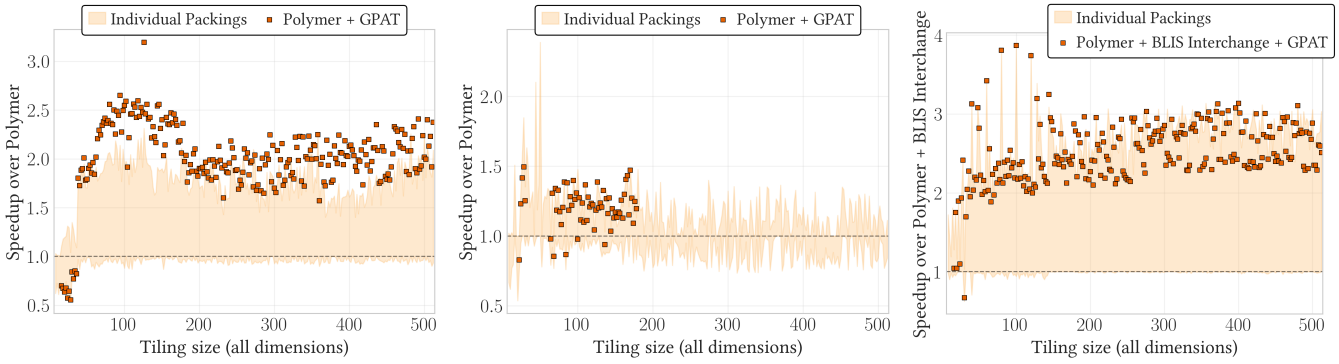
This experiment shows the execution time of **GPAT** and **Individual Packings** relative to **Polymer** to address ①. For each tile size, the performance range from the lowest to the highest performing **Individual Packings** is shown as the light orange area in the graphs. The higher end of the ranges in Figures 1a and 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.

4.3.1 ***2mm.*** Figure 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 $D_{M\times L}$ += $T_{M\times N} * C_{N\times N}$. 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 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 loop unrolling is enabled at the lower tiling sizes, there is a higher frequency of memory spills due to increased register pressure. 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.

4.3.2 ***gemm.*** Figure 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 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 1b, the orange squares of **GPAT** are all within the range of **Individual Packings** because GPAT selected only one candidate.

4.3.3 ***gemm* with BLIS Loop Ordering.** Figure 1c reports on the GEMM benchmark using the loop nest ordering suggested by the BLIS framework [29] — 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.* [9], 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 1c. GPAT found data-layout transformation opportunities that result in simpler memory access patterns — by reducing strides of



(a) *2mm* **(b)** *gemm* **(c)** *gemm* with BLIS Loop Ordering

Figure 1. Packing selection evaluation graphs

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 1b and Figure 1c highlight the importance of building a modular packing compiler pass that is agnostic to optimization decisions made by other compiler passes, and thus indicate an affirmative answer to ③.

4.3.4 TLB Misses and Vectorization. To address ⑤, hardware performance counters were collected on the *2mm* benchmark from the experiment outlined in Section 4.3.1. 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; and (ii) total instruction count, shown in Figure 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 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 4.3.1. GPAT selects these candidates in most tile sizes, leading to the instruction count reduction shown in Figure 3.

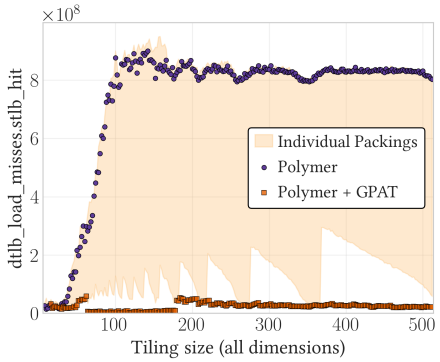


Figure 2. *2mm* L1 dTLB load misses.

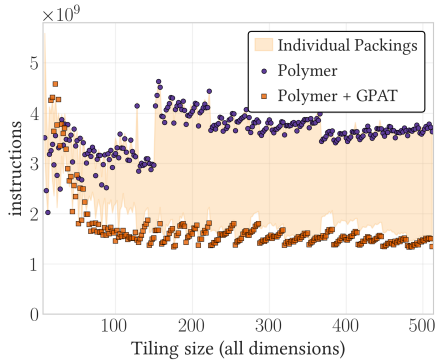


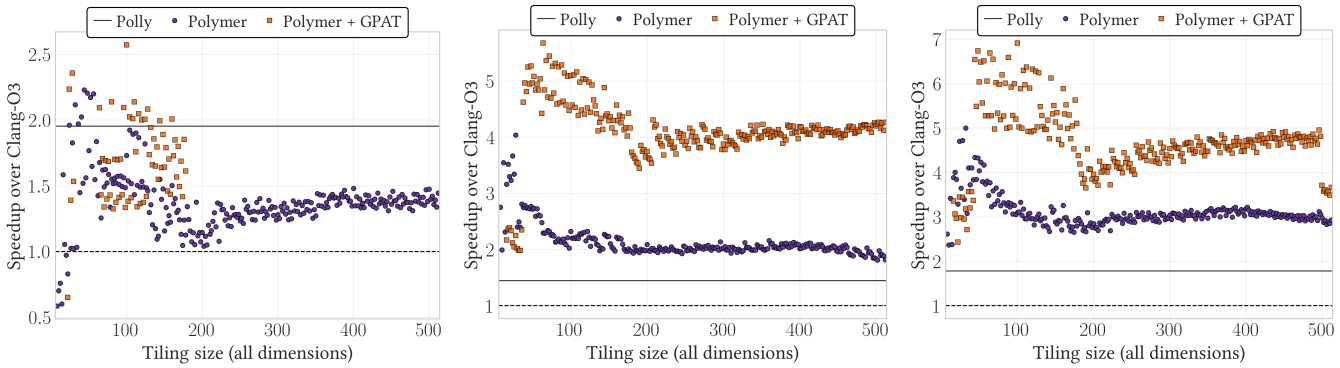
Figure 3. *2mm* instructions count.

Together, Figures 2 and 3 indicate an affirmative answer to ⑤: GPAT can reduce TLB misses and improve vectorization.

4.4 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 after each of the two tiling engines.



(a) *gemm*, baseline time: 446 ms **(b)** *2mm*, baseline time: 1673 ms **(c)** *3mm*, baseline time: 3140 ms

Figure 4. Polybench evaluation on the Polymer tiling engine

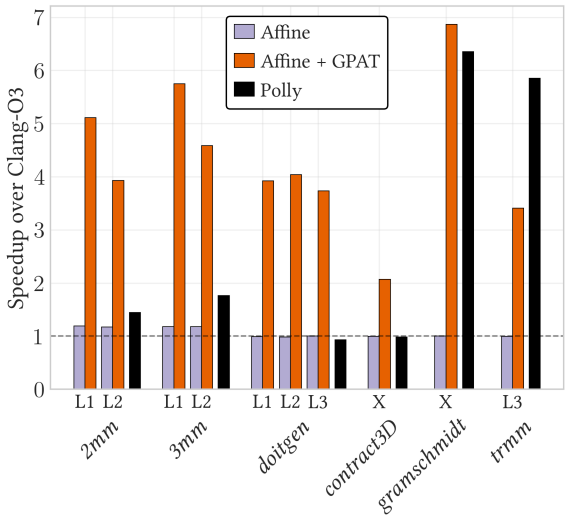


Figure 5. Polybench evaluation on the Affine tiling engine. Baseline times (ms) of each benchmark in the same order of figure: 1673, 3121, 482, 93, 13919, and 2966

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 showing **GPAT** outperforming **Polly** are evidence of GPAT's effectiveness.

4.4.1 Polymer Tiling Engine. As shown in Figure 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 4.3.1). The code generated 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 4.3.2).

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 4.3.2 and *2mm* from Section 4.3.1 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.

4.4.2 Affine Tiling Engine. Figure 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 3.3.

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.

5 Related Works

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 [16]. Nevertheless, as discussed by Lam *et al.*, and later by Temam *et al.* [28], 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 [28]. In the same year, Esseghir proposed a Tile-and-Copy algorithm that couples tiling and copying [7]. 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 [5]. 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 4.3.1). 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 4.3.2).

On modern CPUs, cache interference became less significant when compared to TLB misses due to higher cache associativity and larger problem sizes [27]. Park *et al.* acknowledge the importance of TLB misses and present a mathematical model to estimate TLB utilization in blocked algorithms [23]. 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 [13, 14]. 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 **Basic Linear Algebra Subroutines** (BLAS) libraries [12, 29, 32]. 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.* [19]. 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 [3]. 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 [1]. 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 [20, 30].

6 Conclusion

This work 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. This paper shows that, 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.

Acknowledgments

This research is partially supported by funding from the Natural Sciences and Engineering Research Council of Canada, and by funding from industrial collaborations with IBM Canada and Huawei Technologies Co. Ltd.

A Artifact Appendix

A.1 Abstract

This artifact provides a Docker image containing all required binaries and instructions to execute the two experiments presented in the paper. Additionally, the artifact contains the source code, scripts, benchmarks, and the data and graphs from the presented experiments.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** A generalized packing analysis and code transformation.
- **Program:** Includes source for Polybench/C 4.2 and the running example (Listing 1).
- **Compilation:** Includes LLVM 14.0.1.
- **Transformations:** Includes Polygeist and Polymer at versions specified in Sections 4.1 and 4.2.
- **Binary:** Includes binaries in the Docker image.
- **Run-time environment:** Tested on Ubuntu 20.04.1 (Kernel 5.15.0-46) with Docker. Root access is needed to enable profiling with `perf`.
- **Hardware:** Tested on Intel x86-64 CPUs. Sudo access is required to lock CPU frequency.
- **Metrics:** Execution time, Linux `perf` counters (L1, L2, and L3 cache misses; L1 and L2 TLB misses.),
- **Output:** Execution logs and associated graphs for the experiments. The expected results are also included.
- **Experiments:** Packing selection evaluation (Section 4.3) and Polybench evaluation (Section 4.4).
- **How much disk space required (approximately)?:** 4.1 GiB for the base Docker container. An additional 1.4 GiB for the source code, scripts, and results from the paper.
- **How much time is needed to complete experiments (approximately)?:** 3-4 days, but highly dependent on the CPU and locked frequency.
- **Publicly available?:** Yes.
- **Code licenses?:** Apache License v2.0 with LLVM Exceptions.
- **Archived (provide DOI)?:** 10.5281/zenodo.7517506

A.3 Description

A.3.1 How to Access. The artifact is archived: 10.5281/zenodo.7517506 and the space required is about 5.5 GiB [26].

A.3.2 Hardware Dependencies. A CPU with access to hardware counters for cache misses, TLB misses, and instructions executed.

A.3.3 Software Dependencies. Working Docker installation and Linux `perf`.

A.4 Installation

A.4.1 Perf on Host. To enable `perf`, install it and allow unprivileged users to see `perf` event counters:

```
$ sudo apt install linux-tools-common linux-tools-generic \
    linux-tools-$(uname -r)
$ sudo sh -c 'echo 1 > /proc/sys/kernel/perf_event_paranoid'
```

A.4.2 CPU Scaling. To obtain more consistent and reliable results, lock the CPU frequency. For the experiments presented in this paper, the CPU was locked at its base frequency (3 GHz), which varies across CPUs.

```
$ sudo cpupower frequency-set --governor performance
$ sudo cpupower frequency-set -u 3GHz
$ sudo cpupower frequency-set -d 3GHz
```

A.4.3 Docker Container. Download the Docker image, navigate to the directory containing the download to load the image, and then create and start a container.

```
$ docker load --input docker-packing-artifact.tar.gz
$ docker create --privileged -it --name artifact \
    packing-artifact
$ docker start artifact
```

The `--privileged` flag is needed to use `perf` in the experiments. Additionally, `perf` requires installing the version that matches your system in the container by running:

```
$ docker exec -u 0 -it artifact bash
$ apt update && apt install -y linux-tools-$(uname -r)
$ exit
```

A.5 Experiment Workflow

With the docker container already started, run the Docker container interactively using the following command.

```
$ docker exec -it artifact bash
```

Start by setting up the architectural-specific parameters of the target CPU by modifying *spec.file* inside the container. This can be done, for example with the following command (`vim` or `nano`):

```
$ vim $HOME/scripts/experiments/spec.file
```

The parameters entered in *spec.file* are used by the packing analysis and by LLVM's Polly. They also define what are the cache parameters used as inputs to Affine Tiling in the Polybench evaluation. In the host, cache and TLB information for an x86 machine can be found by running:

```
$ lscpu -C
$ sudo apt install -y x86info && x86info -c
```

A.5.1 Replicate Experiments. To exactly replicate the methods used to obtain the results in this paper, run the following in the container:

```
$ mkdir $HOME/replica
$ $HOME/scripts/experiments/auto-eval.sh $HOME/replica
```

Finally, in the host, retrieve the results by running:

```
$ ID="$(docker ps -aqf 'name=^artifact$')"
$ docker cp ${ID}:/home/packing/replica/ .
```

A.5.2 Detailed Workflow. The following instructions provide a more modular way to run the same experiments. They also allow running the experiment without `perf`, thus not requiring root permissions to execute.

To run the packing selection evaluation on *gemm*, execute the following commands inside the Docker container:

```
$ BENCHMARK="gemm"
$ DATASET_SIZE="LARGE"
$ cd $HOME/scripts/experiments/packing-selection-evaluation/
$ OUTPUT_DIR="$HOME/output/output-${BENCHMARK}-${
    DATASET_SIZE}"
$ mkdir -p ${OUTPUT_DIR}/graphs
$ ./generate-files.sh -D ${DATASET_SIZE} -B ${BENCHMARK} \
    ${OUTPUT_DIR}
$ ./run.sh -D ${DATASET_SIZE} ${OUTPUT_DIR}/executables \
    ${OUTPUT_DIR}
$ ./parse-log.py ${OUTPUT_DIR}/output.log \
    ${OUTPUT_DIR}/graphs ${BENCHMARK}
```

To run the same experiment with *2mm* or *gemm* in the BLIS loop order, change the `BENCHMARK` variable to `"2mm"` or `"gemm-blis"` and rerun the previous commands.

To run the Polybench evaluation experiment with the Polymer tiling engine, execute the following commands inside the Docker container:

```
$ TILING="Polymer"
$ DATASET_SIZE="LARGE"
$ cd $HOME/scripts/experiments/polybench-evaluation/
$ OUTPUT_DIR="$HOME/output/output-polybench-${TILING}-${
    DATASET_SIZE}"
$ mkdir -p ${OUTPUT_DIR}/logs ${OUTPUT_DIR}/graphs
$ ./generate-files.sh -D ${DATASET_SIZE} -T ${TILING} \
    ${OUTPUT_DIR}
$ ./run.sh -D LARGE ${OUTPUT_DIR} ${OUTPUT_DIR}/logs
$ ./parse-log.py ${OUTPUT_DIR}/logs ${OUTPUT_DIR}/graphs \
    ${TILING}
```

To run the same experiment with the Affine Tiling engine, simply change the `TILING` variable to `"AffineTiling"`.

All scripts have a help message, accessed through the `-h` flag, describing the script's use and optional parameters. There are also *README.md* files in the `~/scripts` directory that provide more information.

To collect event counters with `perf` during benchmark execution, add the flag `-p` to the *run.sh* scripts. Additionally, to specify the number of runs of each benchmark, use the flag `-r NUMBER` in the *run.sh* scripts. Lastly, Polybench defines 5 dataset sizes: `EXTRALARGE`, `LARGE`, `MEDIUM`, `SMALL`, and `MINI`. The experiments were run with the `LARGE` dataset. To use a different size, change the variable `DATASET_SIZE` to the desired dataset size.

To copy an experiment's output from the Docker container to the host, run from the host machine:

```
$ ID="$(docker ps -aqf 'name=^artifact$')"
$ docker cp ${ID}:/home/packing/output/ .
```

A.6 Evaluation and Expected Results

If using a similar environment, the result trends should follow what was presented in this paper. However, a different CPU may have different features, affecting the end results. For example, large changes to the cache and TLB sizes, or support for additional instructions may affect the final results.

A.7 Experiment Customization

The scripts allow selecting the Polybench dataset size used in the experiments. Further, in the experiment's *spec.file*, there are three false-by-default variables that can be set to true.

- `POLLY_ENABLE_PATTERN_MATCHING`: enables pattern-match-based optimizations in Polly, affecting all Polly compiled benchmarks when matrix multiplication can be detected.
- `LLVM_DISABLE_VECTORIZATION`: disables vectorization passes in LLVM, affecting all compiled benchmarks.
- `LLVM_DISABLE_UNROLLING`: disables unrolling passes in LLVM, affecting all compiled benchmarks.

To customize the Polybench evaluation workflow, scripts are available in `~/scripts` in folders named: affine-tiling, lower-to-binary, mlir-packing, polly, polygeist, and polymer.

Interacting with these scripts is not necessary to execute the experiments, but they provide more details on the compilation pipeline used. Each folder contains a script with a help message and a *README.md* file.

A.8 Notes

- Depending on the host's Docker configuration, `sudo` privileges may be required.
- Some tiling sizes cause MLIR to generate incorrect code for *trmm* leading to failures. However, these failures do not affect the other benchmarks.
- In the Polybench evaluation using the Affine tiling engine, the summary graph (Figure 5) is sensitive to hardware parameter changes and may fail to render correctly. Please refer to the graphs that show data for benchmarks individually in this case.
- A host machine not using Ubuntu nor an Intel CPU may require changing the `perf` event counter names in the `run.sh` scripts. If the `perf` container installation fails try the following command before using `perf`.

```
$ export PATH=/usr/lib/linux-tools/5.4.0-132-generic/:$PATH
```

References

[1] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. arXiv:2003.00532

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computer Machinery (ACM), New York, NY, USA, 101–113. https://doi.org/10.1145/1379022.1375595

[3] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. 2019. TSM2: Optimizing Tall-and-Skinny Matrix-Matrix Multiplication on GPUs. In *Proceedings of the 33rd ACM International Conference on Supercomputing* (Phoenix, Arizona, USA) *(ICS 2019)*. Association for Computer Machinery (ACM), New York, NY, USA, 106–116. https://doi.org/10.1145/3330345.3330355

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI 2018)*. USENIX Association, Berkley, CA, USA, 579–594. https://doi.org/10.48550/arXiv.1802.04799

[5] Stephanie Coleman and Kathryn S. McKinley. 1995. Tile Size Selection Using Cache Organization and Data Layout. *ACM SIGPLAN Notices* 30, 6 (June 1995), 279–290. https://doi.org/10.1145/207110.207162

[6] João P. L. De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. 2021. KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls. *ACM Transactions on Architecture and Code Optimization* 18, 3 (Sept. 2021), 1–22. https://doi.org/10.1145/3459010

[7] Karim Esseghir. 1993. *Improving Data Locality for Caches*. Master's thesis. Rice University.

[8] Google. 2013. Benchmark. https://github.com/google/benchmark

[9] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Software* 34, 3 (May 2008), 1–25. https://doi.org/10.1145/1356052.1356053

[10] Kazushige Goto and Robert Van De Geijn. 2008. High-Performance Implementation of the Level-3 BLAS. *ACM Trans. Math. Software* 35, 1 (July 2008), 14 pages. https://doi.org/10.1145/1377603.1377607

[11] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 4 (Dec. 2012). https://doi.org/10.1142/S0129626412500107

[12] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. Retrieved August 22, 2022 from http://eigen.tuxfamily.org

[13] Ismail Kadayif and Mahmut Kandemir. 2004. Quasidynamic Layout Optimizations for Improving Data Locality. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 15, 11 (Nov. 2004), 996–1011. https://doi.org/10.1109/TPDS.2004.70

[14] Mahmut Kandemir, Alok Choudhary, Jagannathan Ramanujam, and Prith Banerjee. 1998. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (Dallas, Texas, USA) *(MICRO 31)*. IEEE Computer Society Press, Washington, DC, USA, 285–296.

[15] Aditya Kumar and Sebastian Pop. 2016. SCoP Detection: A Fast Algorithm for Industrial Compilers. In *Proceedings of the 6th International Workshop on Polyhedral Compilation Techniques* (Prague, Czech Republic) *(IMPACT 2016)*. Association for Computer Machinery (ACM), New York, NY, USA.

[16] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. *ACM SIGPLAN Notices* 26, 4 (April 1991), 63–74. https://doi.org/10.1145/106973.106981

[17] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd ACM/IEEE International Symposium on Code Generation and Optimization* (Palo Alto, California, USA) *(CGO 2004)*. IEEE Computer Society Press, Washington, DC, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[18] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the 19th ACM/IEEE International Symposium on Code Generation and Optimization* (Seoul, Republic of Korea) *(CGO 2021)*. IEEE Computer Society Press, Washington, DC, USA, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[19] Chendi Li, Haipeng Jia, Hang Cao, Jianyu Yao, Boqian Shi, Chunyang Xiang, Jinbo Sun, Pengqi Lu, and Yunquan Zhang. 2021. AutoTSMM: An Auto-tuning Framework for Building High-Performance Tall-and-Skinny Matrix-Matrix Multiplication on CPUs. In *Proceedings of the 2021 IEEE 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, Washington, DC, USA, 159–166. https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00034

[20] Jintao Meng, Chen Zhuang, Peng Chen, Mohamed Wahib, Bertil Schmidt, Xiao Wang, Haidong Lan, Dou Wu, Minwen Deng, Yanjie Wei, and Shengzhong Feng. 2022. Automatic Generation of High-Performance Convolution Kernels on ARM CPUs for Deep Learning. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 33, 11 (Nov. 2022), 2885–2899. https://doi.org/10.1109/TPDS.2022.3146257

[21] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, and Karin Hogstedt. 1998. Quantifying the Multi-Level Nature of Tiling Interactions. *International Journal of Parallel Programming* 26, 6 (Dec. 1998), 641–670. https://doi.org/10.1023/A:1018782528453

[22] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the 30th ACM International Conference on Parallel Architectures and Compilation Techniques* (Atlanta, GA, USA) *(PACT 2021)*. IEEE Computer Society Press, Washington, DC, USA, 45–59. https://doi.org/10.1109/PACT52795.2021.00011

[23] N. Park, B. Hong, and V.K. Prasanna. 2003. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 14, 7 (July 2003), 640–654. https://doi.org/10.1109/TPDS.2003.1214317

[24] Louis-Noel Pouchet and Tomofumi Yuki. 2016. Polybench: The polyhedral benchmark suite (version 4.2.1). Retrieved August 22, 2022 from https://sourceforge.net/projects/polybench/

[25] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM International Conference on Parallel Architectures and Compilation Techniques* (Seattle, Washington, USA) *(PLDI 2013)*. Association for Computer Machinery (ACM), New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[26] Caio Salvador Rohwedder, Nathan Henderson, João P. L. de Carvalho, Yufei Chen, and J. Nelson Amaral. 2023. Artifact of "To Pack or Not to Pack: A Generalized Packing Analysis and Transformation". Zenodo. https://doi.org/10.5281/zenodo.7517506

[27] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. 2000. Recency-Based TLB Preloading. In *ACM SIGARCH Computer Architecture News* (Vancouver, British Columbia, Canada) *(ISCA '00)*. Association for Computer Machinery (ACM), New York, NY, USA, 117–127. https://doi.org/10.1145/342001.339666

[28] Olivier Temam, Elana D. Granston, and William Jalby. 1993. To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Proceedings of the 1993 ACM/IEEE Supercomputing Conference* (Portland, Oregon, USA) *(Supercomputing 1993)*. Association for Computer Machinery (ACM), Portland, OR, USA, 410–419. https://doi.org/10.1145/169627.

169762

[29] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3 (June 2015), 1–33. https://doi.org/10.1145/2764454

[30] Bram Wasti, José Pablo Cambronero, Benoit Steiner, Hugh Leather, and Aleksandar Zlateski. 2020. LoopStack: a Lightweight Tensor Algebra Compiler Stack. arXiv:2205.00618

[31] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor, and Mary Hall. 2020. Autotuning PolyBench Benchmarks with LLVM Clang/Polly Loop Optimization Pragmas Using Bayesian Optimization. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems* (GA, USA) *(PMBS)*. IEEE Computer Society Press, Washington, DC, USA, 61–70. https://doi.org/10.1109/PMBS51919.2020.00012

[32] Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems* (Singapore) *(ICPADS 2012)*. IEEE Computer Society Press, Washington, DC, USA, 684–691. https://doi.org/10.1109/ICPADS.2012.97

Received 2022-09-02; accepted 2022-11-07