Performance-Portable Tensor Transpositions in MLIR

Mahesh Lakshminarasimhan, Mary Hall, and P. Sadayappan

Kahlert School of Computing, University of Utah, USA
{maheshl,mhall,saday}@cs.utah.edu

Abstract. This paper presents an optimized code generation for arbitrary out-of-place tensor transpositions using the MLIR compiler infrastructure, portable across CPU architectures. The proposed modular and reusable approach encodes optimizations such as multi-level tiling and explicit vectorization at multiple levels of abstraction as a sequence of transformation and conversion passes in MLIR. The efficient code generated is evaluated on AMD, Intel, and ARM processors and achieves performance comparable to the state-of-the-art HPTT library [27], a compelling speedup over Eigen [10], and a significant fraction of the STREAM memory bandwidth on these platforms. We further integrate this progressive lowering pipeline into COMET, an MLIR-based compiler for tensor contractions, and obtain an average speedup of 26%.

Keywords: Multi-Level Intermediate Representation · Tensor Transposition · SIMD architectures.

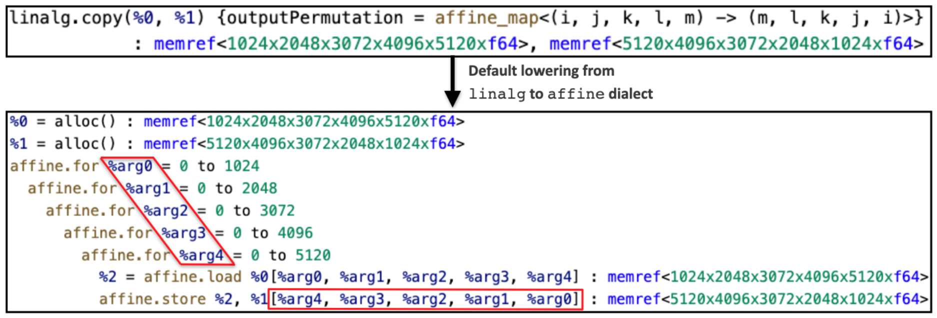
1 Introduction

Tensor transposition is a generalization of matrix transposition for higher dimensions. It is an important data layout transformation primitive for tensor algebra operations in various application domains, including machine learning, computational chemistry, quantum many-body methods, and climate simulations. The tensor transpose operation involves the permutation of indices of a given tensor: $B_{\Pi(i_0,i_1,i_2,\ldots,i_{n-1})} \leftarrow A_{i_0,i_1,i_2,\ldots,i_{n-1}}$, where A and B are input and output tensors respectively, and Π denotes index permutation for the transposition.

Prior work on improving the performance of arbitrary tensor transpositions include highly optimized libraries, notably HPTT (High-Performance Tensor Transpose) [27], and others [30, 16, 18, 10, 21], and source-to-source code generators with autotuning [26, 31]. While they achieve high performance, it is quite challenging to integrate them into general-purpose and domain-specific compilers. It also requires significant engineering effort and expertise to repeatedly port these libraries to similar and future architectures.

Multi-Level Intermediate Representation (MLIR) [19] is a recent compiler framework that facilitates building reusable and extensible compiler infrastructures. MLIR breaks the isolation between domains and enables comprehensive optimizations by expressing the computation at different levels of abstractions called *dialects*. MLIR can be organically used to represent architecture-independent transformations (e.g., tiling) at the higher-level dialects, and platform-specific optimizations (e.g., vectorization) at the lower-level dialects. During compilation, the source program's high-level representation is progressively optimized and transformed to lower-level abstractions, until reaching a low-level, general-purpose representation for code generation. Several domain-specific compilers [2, 5, 28, 3, 7, 4, 1, 9, 17, 15, 23, 6] are under development leveraging the MLIR framework, and many of them [5, 28, 2, 23, 6, 7] involve data layout transformations with tensor transposition at different levels.

Though tensor transpositions have zero arithmetic intensity and exhibit no temporal locality, they pose challenges to modern memory subsystems due to their irregular memory accesses and potentially large strides. Figure 1 shows the MLIR code lowered from `copy()` operation in `linalg` dialect to the `affine` dialect for an out-of-place 5D tensor transpose ($B_{i_4,i_3,i_2,i_1,i_0} \leftarrow A_{i_0,i_1,i_2,i_3,i_4}$). The naive lowering results in a 5D affine loop nest that reads from the input `memref` and copies it to output `memref` for the corresponding index permutation. This native code exhibits poor spatial locality due to the high-access stride of the output `memref`. To achieve high performance for tensor transpositions, it is essential to effectively exploit the deep cache hierarchies, ample parallelism, and vectorization capabilities of modern CPUs.



```
linalg.copy(%0, %1) {outputPermutation = affine_map<(i, j, k, l, m) -> (m, l, k, j, i)>}
              : memref<1024x2048x3072x4096x5120xf64>, memref<5120x4096x3072x2048x1024xf64>
```

```
%0 = alloc() : memref<1024x2048x3072x4096x5120xf64>
%1 = alloc() : memref<5120x4096x3072x2048x1024xf64>
affine.for %arg0 = 0 to 1024
  affine.for %arg1 = 0 to 2048
    affine.for %arg2 = 0 to 3072
      affine.for %arg3 = 0 to 4096
        affine.for %arg4 = 0 to 5120
          %2 = affine.load %0[%arg0, %arg1, %arg2, %arg3, %arg4] : memref<1024x2048x3072x4096x5120xf64>
          affine.store %2, %1[%arg4, %arg3, %arg2, %arg1, %arg0] : memref<5120x4096x3072x2048x1024xf64>
```

Fig. 1: 5D tensor transpose code lowered from `linalg` to `affine` dialect in MLIR

This necessitates a systematic, modular, and reusable approach for generating tensor transpose kernels by exploring design points and optimizing them efficiently. In this paper, we propose an optimized code generation for arbitrary tensor transpositions by implementing various optimizations at multiple abstraction levels in MLIR. The primary contributions of this work are as follows:

- We develop a progressive transformation and lowering pipeline in MLIR that generates high-performance parallel code for tensor transpositions portable across different CPU architectures (section 2).
- We perform an evaluation of generated code on four different platforms based on AVX, AVX-512, and ARM SVE instruction sets. We assess performance

portability on these architectures using the efficiency metric proposed in [24] (section 3).

– We integrate this code generation pipeline for optimized transpositions into the MLIR-based COMET [23] compiler and demonstrate speedup for tensor contractions (section 4).

2 Design and Implementation

The code generation pursued in this work with MLIR generates kernels for parallel out-of-place tensor transpositions of arbitrary order (dimensionality) and single and double floating-point precisions. It is portable across SIMD architectures with support for AVX2, AVX-512, and ARM SVE, and extensible to other vector instruction sets. Optimizations such as multi-level tiling, loop reordering, multithreading, and explicit vectorization are progressively applied at multiple levels of the IR.

We adapt the key design principles from HPTT [27], the state-of-the-art C++ library for tensor transpositions on CPUs that supports up to AVX2 instructions. Arbitrary tensor transpositions are decomposed into independent 2D transposes, and each 2D slice is further decomposed into two levels of tiles: macrotiles and microtiles. Macrotiles are parallelized over different threads and each microtile is computed by an explicitly vectorized microkernel. Figure 2 shows the reduction of a 3D tensor transposition into a series of independent 2D microtiles.

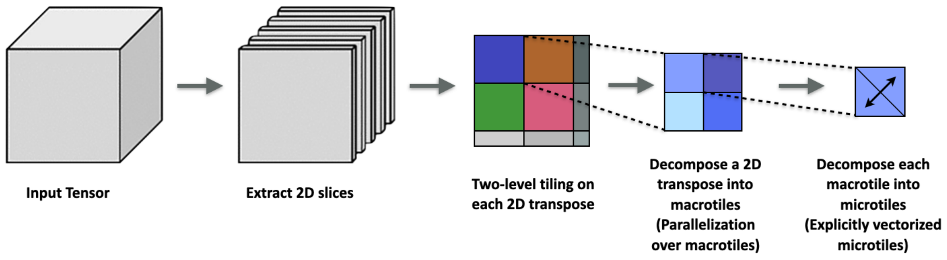

Input Tensor
Extract 2D slices
Two-level tiling on each 2D transpose
Decompose a 2D transpose into macrotiles (Parallelization over macrotiles)
Decompose each macrotile into microtiles (Explicitly vectorized microtiles)

Fig. 2: Reducing a 3D tensor transposition to a series of 2D microtiles.

2.1 Overview of Code Generation in MLIR

We implement a progressive lowering pipeline that encodes optimizations at multiple dialects in MLIR, which is shown in figure 3.

Entry Point for Code Generation We begin the code generation at the `linalg` dialect using the `copy` operation as an abstraction for tensor transpositions. The `linalg.copy()` operation copies data from an input `view` to an output `view` and reorders the indices of the output `view` for a given `permutation` attribute. Figure 1 shows an example of `linalg.copy`. The entry point for code generation can also be an IR, for example, generated from a frontend like TensorFlow lowered to `MHLO` and then to the `linalg` dialect.

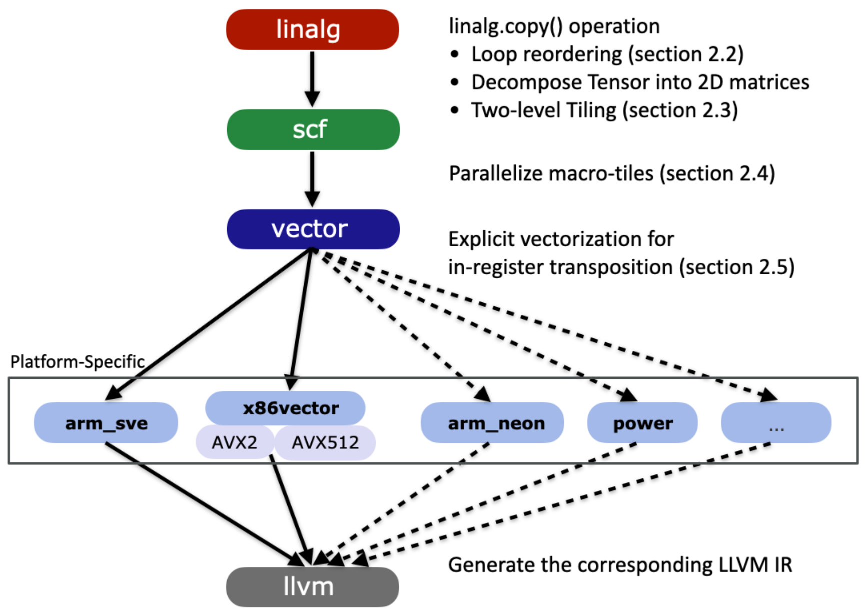

linalg
scf
vector
linalg.copy() operation
• Loop reordering (section 2.2)
• Decompose Tensor into 2D matrices
• Two-level Tiling (section 2.3)
Parallelize macro-tiles (section 2.4)
Explicit vectorization for
in-register transposition (section 2.5)
Platform-Specific
arm_sve
x86vector
AVX2
AVX512
arm_neon
power
...
Generate the corresponding LLVM IR
llvm

Fig. 3: The progressive lowering pipeline for the `linalg.copy` operation to generate the corresponding LLVM IR. Dashed lines indicate lowering passes for future extension.

Transformations and Lowering in MLIR The initial step is to reorder the loops for the `linalg.copy` operation at an optimal order (section 2.2). Two-dimensional slices are then extracted from the input tensor using the `extract_slice` operation. We implement a custom transformation pass to perform two-level tiling that decomposes each of the extracted 2D slice into macrotiles and microtiles (section 2.3). The next step is lowering to the `scf` dialect where the loops corresponding to macrotiles are parallelized (section 2.4). We then lower to the `vector` dialect and further down to dialects specific to the instruction set – `x86vector` (for AVX and AVX-512) and `arm_sve`. At this stage, the microtiles are computed by architecture-specific, hand-optimized microkernels that are explicitly vectorized and perform an in-register two-way transposition (section 2.5). The final step is to lower to the `llvm` dialect and generate the LLVM IR corresponding to the lowered vector code. For a given entry point (e.g., `linalg.copy`), the end-to-end code generation process is automated.

Portability The ability to progressively apply optimizations in multiple dialects of MLIR simplifies the generation of code portable to different platforms. The lowering pipeline in figure 3 is architecture-independent until the `vector` dialect and the subsequent lowering to platform-specific dialects generates vectorized code corresponding to the vector instruction set. This makes code generation extensible to different CPU architectures, with the microkernel alone needing to be implemented, and the existing pipeline can be reused. We currently support code generation for x86 and ARM SVE-based CPUs, and this approach can be simply extended to ARM Neoverse, IBM Power, and other instruction sets by implementing the corresponding microkernel.

2.2 Loop reordering

An n-dimensional tensor transpose has $n!$ distinct orderings of loops. The choice of best loop order for an n-way transpose has a significant impact on its performance, as observed in [13]. The optimal loop order is one that maximizes the data locality of both input and output tensors, resulting in reduced cache and TLB misses. This also minimizes the access stride within the innermost loop, which helps the hardware prefetchers better learn the memory access patterns. Hence, placing the loops corresponding to the fastest varying (stride-1) indices of both input and output tensors among the innermost levels of the loop yields high performance. For example, the best loop order for the transposition in figure 1 $B_{i_4,i_3,i_2,i_1,i_0} \leftarrow A_{i_0,i_1,i_2,i_3,i_4}$ is $[i_2, i_1, i_3, i_0, i_4]$, where i_2 and i_4 are the outermost and innermost loops respectively.

We determine the optimal loop order for an arbitrary transposition using a heuristic-based cost model. The model assigns a cost to each loop index based on its position with respect to both input and output tensors, with the cost exponentially increasing from the innermost to the outermost loop. The overall cost of the loop order is computed by summing up the cost of each loop index. The overall cost is the least when the innermost indices of both input and output tensors are the innermost loops. The optimal loop order obtained from this model is set using the `mlir::interchangeLoops` transformation utility.

It needs to be noted that this model does not account for the extent of tensor modes to determine the best loop order. For certain higher dimensional tensor transpositions, it may be more performant to place the modes with shorter extents at the innermost loop, instead of fastest varying modes. Determining this requires a more sophisticated analytical model, which is out of the scope of this paper.

2.3 Multi-Level Tiling

To further increase spatial locality in both input and output tensors and facilitate vectorization, it is essential to restructure memory accesses by performing two-level tiling. Two-dimensional slices extracted from the input tensor are decomposed into independent macrotiles and microtiles. These tiles preserve the stride-1 access of the input and output tensors, since the loop order is set to have the innermost indices of both tensors at the innermost loop levels. The macrotiles are executed in parallel with multithreading (section 2.4), and the microtiles are computed by a fully vectorized microkernel (section 2.5).

The size of the microtile μ is set as the SIMD width of the underlying architecture (e.g., $\mu = 8$ for double precision tensors on AVX-512-based architectures). The macrotile size ρ is fixed to 4 times that of microtile ($\rho = 4.\mu$). This enables data to be moved at the granularity of cachelines and also helps hardware prefetchers fetch adjacent cachelines. This also reduces false sharing of cachelines among threads when macrotiles are executed in parallel.

For irregularly shaped input tensors when its stride-1 extent is not divisible by macrotile size ρ, the size of partial macrotiles is altered accordingly to $\rho = 2.\mu$ or $\rho = \mu$ for utilizing vectorized microtiles. In cases when the partial tiles are smaller than μ, the remainder is computed by a non-vectorized generic transpose kernel.

We implement a custom transformation pass in the `linalg` dialect to tile the `copy` kernel in two levels to generate macrotiles and microtiles, and also to handle partial tiles.

2.4 Parallelization

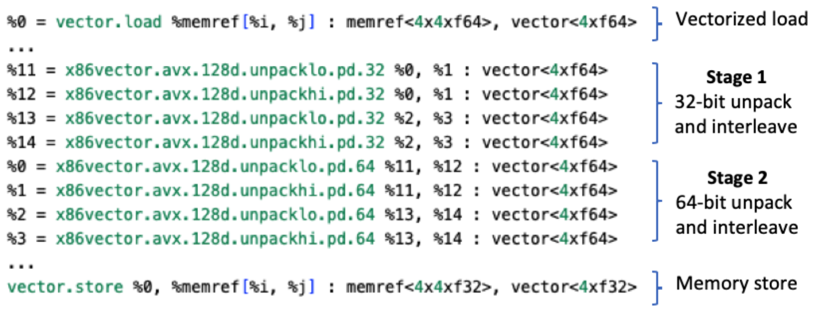
After tiling in the `linalg` dialect, we then lower to the `scf` dialect. The generated macrotiles are entirely independent of each other and can be executed in parallel by different threads. The `scf.for` loops corresponding to macrotiles to be parallelized are marked and converted to `scf.parallel`. Parallelization of the innermost loops (fastest varying indices) is avoided to eliminate highly strided memory accesses and false sharing among threads.

2.5 Explicit Vectorization

The `scf` dialect is now lowered to the `vector` dialect using the `vector.transpose` abstraction. Each microtile is computed by an explicitly vectorized microkernel after lowering to ISA-specific dialects – `x86vector` and `arm_sve`. Each microkernel for single and double-precision elements is manually implemented using vector intrinsics in AVX2, AVX-512, and ARM SVE (in this work, the vector width for SVE is fixed to 512). We added new MLIR operations for vector instrinsics that did not exist in `x86vector` and `arm_sve` dialects to implement the microkernels.

The microkernel performs an in-register transposition of each $\mu \times \mu$ microtile in $log_2\mu$ number of steps. Figure 4 shows an exemplary microkernel for a 4×4 microtile implemented with AVX instructions that get completed in $log_2 4 = 2$ steps. Initially, the 16 microtile elements are loaded from memory into registers using vectorized loads. Then, in two steps, in-register transposition is performed by unpacking and interleaving the lower and higher halves of registers. Finally, the transposed microtile is stored back in memory.

As another example, a 16×16 microtile with single-precision elements can transposed in $log_2 16 = 4$ steps using AVX-512 instructions in a total of 64 cycles.



```
%0 = vector.load %memref[%i, %j] : memref<4x4xf64>, vector<4xf64>
...
%11 = x86vector.avx.128d.unpacklo.pd.32 %0, %1 : vector<4xf64>
%12 = x86vector.avx.128d.unpackhi.pd.32 %0, %1 : vector<4xf64>
%13 = x86vector.avx.128d.unpacklo.pd.32 %2, %3 : vector<4xf64>
%14 = x86vector.avx.128d.unpackhi.pd.32 %2, %3 : vector<4xf64>
%0 = x86vector.avx.128d.unpacklo.pd.64 %11, %12 : vector<4xf64>
%1 = x86vector.avx.128d.unpackhi.pd.64 %11, %12 : vector<4xf64>
%2 = x86vector.avx.128d.unpacklo.pd.64 %13, %14 : vector<4xf64>
%3 = x86vector.avx.128d.unpackhi.pd.64 %13, %14 : vector<4xf64>
...
vector.store %0, %memref[%i, %j] : memref<4x4xf32>, vector<4xf32>
```

Fig. 4: Pseudocode for an exemplar microkernel with AVX vector intrinsics for a 4×4 double-precision microtile.

In each step, the shuffling and interleaving width of logical elements is expanded. We also increment the distance at which we access the 32 registers grouped into two sets of 16 registers each. As in the previous example, the first two steps respectively involve 32-bit and 64-bit interleaves of register elements, resulting in a pair of output registers holding intermediate transpose elements at 128-bit granularity. The third step is shuffling register pairs at 128-bit granularity, which outputs registers holding 256-bit of transposed data. The fourth and final step is to shuffle these 256-bit registers again, resulting in a final set of 16 registers holding the 16×16 transposed microtile.

We adopt this blueprint for implementing microkernels instead of HPTT's since the HPTT microkernel involves the use of local buffers and causes additional data movement. Our implementation enables us to extend the microkernel to architectures with wider SIMD lengths, and perform in-register transpositions efficiently, as presented in the above examples.

For the case of remainder partial tiles (discussed earlier in section 2.3), we currently use a generic non-vectorized transpose kernel. Instead, partial tiles could also be fully vectorized by employing masked loads and stores in the microkernel, and further improve the performance. Moreover, we could also automatically generate the microkernel while lowering to ISA-specific dialects for a more simplified end-to-end code generation. Both of these extensions are the subject of future work.

3 Performance Evaluation

In this section, we compare the performance of tensor transpose kernels generated with our approach against state-of-the-art libraries on four different architectures. We also use an efficiency metric [24] to evaluate the performance portability of our code generation.

3.1 Experimental Setup

Hardware Description We evaluate the achieved performance on four systems: (i) LBNL Cori's Intel Haswell (results were collected before its decommissioning in May 2023), (ii) ORNL's Frontier AMD CPU, (iii) Intel Ice Lake from University of Utah's Notchpeak cluster, and (iv) Stony Brook's Ookami with Fujitsu A64FX processor, which is a prototype of the Fugaku supercomputer. (i) and (ii) support up to AVX2 vector instructions, (iii) has AVX-512 enabled, and (iv) is based on ARM SVE (the vector width is set to 512). Table 1 provides details of the four platforms evaluated.

Table 1: Evaluated hardware platforms

System	**Processor**	**Micro-architecture**	**Instruction Set Extensions**	**#Threads**	**STREAM Bandwidth (GB/s)**
Cori	Intel Xeon E5-2698 v3	Haswell	AVX2	56	65
Frontier	AMD EPYC Trento 7A53	Zen 3	AVX2	128	158
Notchpeak	Intel Xeon Gold 6330	Ice Lake	AVX-512	112	177
Ookami	Fujitsu A64FX	ARMv8.2-A	ARM SVE	48	118

Baselines The optimized code generated with our approach is compared against:

- High-Performance Tensor Transposition (HPTT) C++ library [27] set up to perform tensor transposes without scaling and accumulating the inputs, with all optimizations turned on, and autotuning enabled to determine the best loop permutation and parallelization strategy.
- Eigen [10] C++ template library (v3.4.0).
- STREAM triad benchmark [22], to estimate the execution efficiency of the generated code in terms of achieved memory bandwidth.

Dataset We use a set of 57 single and double-precision tensor transpositions from the TTC benchmark [26] ranging from 2D to 6D, with each tensor occupying about 200 MB of memory, which is larger than the last level cache of the systems evaluated. The dataset comprises test cases of inverse transposes, and cases where the stride-1 retains its position after transposition. It also includes transposes where the extent of all tensor modes are kept the same, and cases where the largest mode is substantially bigger than the smallest mode of the tensor. This variety of test cases provides broad diversity for performance analysis, and we thus choose this dataset for evaluation.

The achieved memory bandwidth is calculated as $\frac{2 \times S \times D}{10^9 \times time}$, given the size S of the tensor (product of its dimension sizes), and $time$ taken in seconds for its transposition. The value of D is 4 and 8 for single and double-precision tensors, respectively. Each test case is run ten times and the average is reported with caches cleared after each run.

3.2 Performance Relative to HPTT on AVX-Based CPUs

Figure 5 compares the performance of our transpose code generated against HPTT and Eigen on Cori's Haswell and Frontier's AMD CPUs for double-precision tensors. Over Eigen, we achieve a maximum speedup of 2.7x and an average speedup of 1.9x on Cori and a maximum speedup of 3.1x and an average speedup of 2.3x on Frontier. On average, we obtain a significant fraction of 80% and 78% of the STREAM memory bandwidth on Cori and Frontier, respectively.

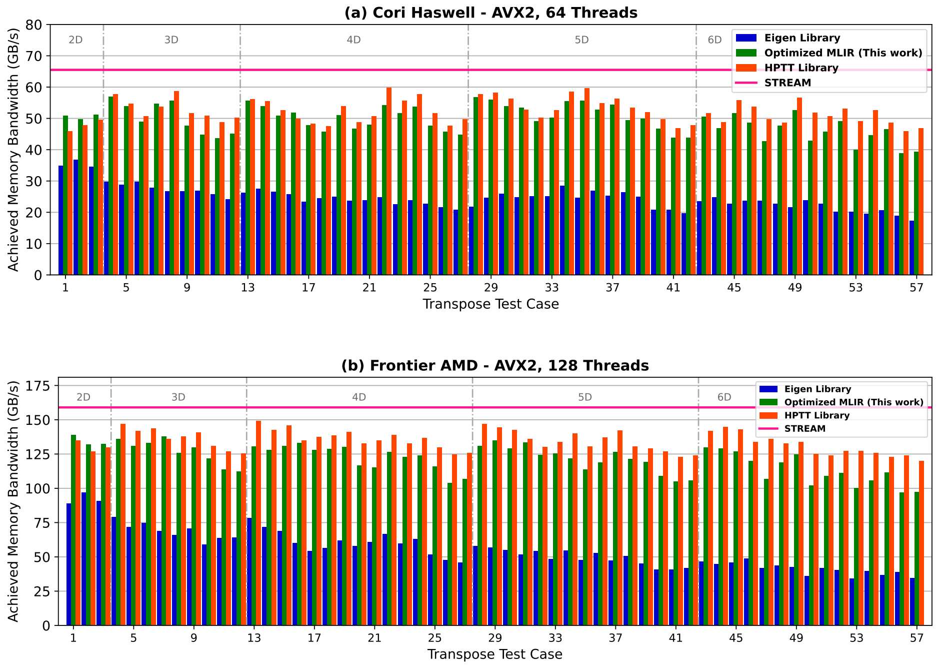


Fig. 5: Performance comparison with HPTT and Eigen libraries on AVX2-enabled Cori and Frontier systems.

We achieve performance comparable to HPTT despite the additional optimizations performed by it that include software prefetching and employing non-temporal (streaming) stores, along with autotuning. We attain an average of 92% and 88% of HPTT's performance, respectively, on Cori and Frontier. For 2D transpositions, our approach slightly outperforms HPTT. For certain higher dimensional transpositions where the stride-1 is identical in both tensors and when the extent of tensor modes is similar, we perform on par with

HPTT, implying that the additional optimizations by HPTT are not very beneficial. In some cases, HPTT parallelizes the stride-1 indices as well to perform load balancing, which leads to non-consecutive memory accesses in threads. In a few other cases, the non-temporal stores set by HPTT get overridden by the temporal stores issued by the compiler.

For some 6D transpositions with a large difference between the extent of tensor modes, autotuned HPTT performs much better than our code. This is so because the model used by us for loop reordering does not factor in the extent of the tensor modes. So, for those cases, it could be optimal to place the indices with smaller extents at the innermost loop levels instead of the stride-1 indices.

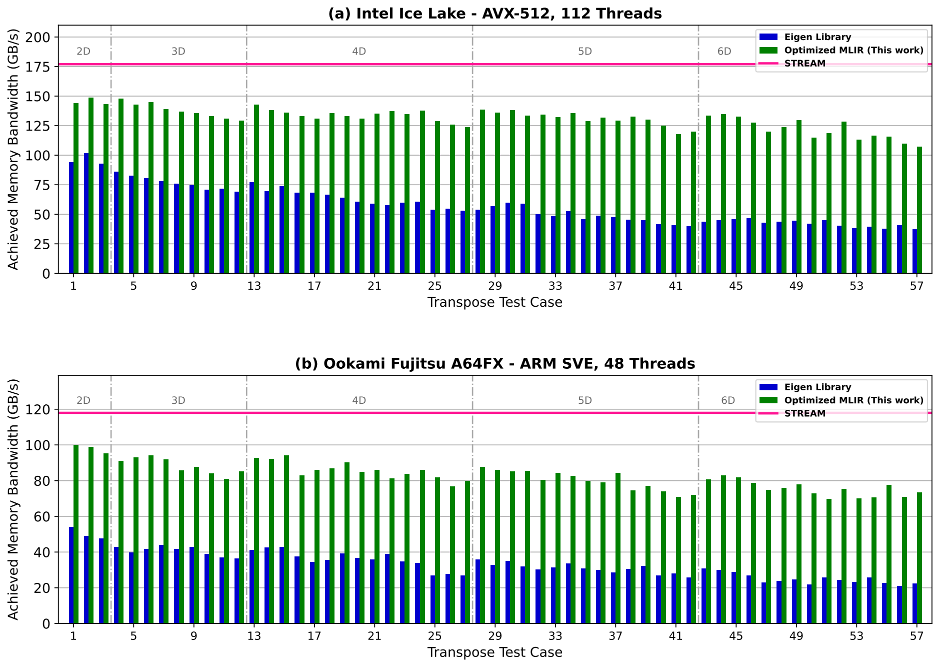


Fig. 6: Performance comparison with Eigen library on Intel Ice Lake (AVX-512) and Fujitsu A64FX (ARM SVE).

3.3 Performance on AVX-512 and ARM SVE Systems

Figure 6 shows the memory bandwidth obtained on Intel Ice Lake and Ookami's Fujitsu A64FX processors for double-precision tensors. Since HPTT supports only up to AVX2 intrinsics, we are unable to evaluate its performance on these systems. We thus compare the achieved performance against the Eigen and

STREAM benchmarks. Our code attains a maximum and average speedup of 2.1 and 2.9, respectively, over Eigen and an average of 75% of STREAM bandwidth on Intel Ice Lake. However, on Ookami, both Eigen and our code do not perform as well as they did on other platforms. Our generated code obtains 72% of STREAM bandwidth, and a remarkable average speedup of 3.1 and a maximum speedup of 3.8 over Eigen.

3.4 Performance Portability Metric

To assess the portability of the pursued code generation approach, we adopt the metric Φ defined by Pennycook et. al. in [24]. The metric Φ is calculated as the harmonic mean of the application's performance efficiency across platforms, as shown in the equation below:

$$\Phi(a, p, H) = \begin{cases} \frac{|H|}{\Sigma_{i \in H} \frac{1}{e_i(a,p)}}, & \text{if } i \text{ is supported, } \forall i \in H \\ 0, & \text{otherwise} \end{cases}$$

where $e_i(a, p)$ is the metric's performance efficiency for application a and problem p on platform i. Consistently high efficiencies produce high performance portability Φ.

For this work, we define the performance efficiency e_i as a fraction of the system's STREAM triad memory bandwidth. Table 2 shows the efficiency of the generated code for single and double-precision tensor transpositions on the four platforms. The overall performance efficiency Φ is 74%, which is a significant fraction of the STREAM memory bandwidth, indicating the portable performance achieved by the code generated for FP32 and FP64 transposes on the four platforms.

Table 2: Application efficiency e_i of double and single-precision tensor transpositions across four platforms as a fraction of STREAM bandwidth.

Precision	**Cori**	**Frontier**	**Ice Lake**	**Ookami**	**Efficiency**
FP64	80%	78%	75%	72%	**76%**
FP32	79%	76%	71%	69%	**73%**
				Overall Φ	**74%**

4 Case Study: Compiler Integration

In this section, we illustrate the application of transpositions in tensor contractions, describe the integration of our transpose code generation into the MLIR-based COMET compiler, and analyze the performance of resulting tensor contractions.

4.1 Background: Tensor Contractions

Tensor contraction is a high-dimension generalization of matrix multiplication that is at the core of various science and engineering applications. For example, $C[a,b,c,d] = \sum_{e,f} A[e,a,f,c] \times B[b,f,d,e]$ is a tensor contraction from the coupled-cluster method in quantum chemistry where two 4D tensors A and B are contracted over indices e and f to output a 4D tensor C. The direct approach to perform tensor contractions is to implement a nested loop. Optimizations on these loop nests, such as tiling, loop fusion, and vectorization, often yield suboptimal performance due to highly strided memory access patterns that result in poor cache locality.

Transpose-Transpose-GEMM-Transpose (TTGT) The indirect TTGT approach performs index permutations of input tensors using explicit tensor transpositions followed by a GEMM operation, and a final permutation of the resulting matrix to reconstruct the output tensor. The transposition of input tensors flattens (or unfolds) them into matrices by reordering their indices and then merging consecutive indices. The TTGT approach is widely adopted due to its conceptual simplicity and generality, and the availability of highly-optimized vendor-provided GEMM routines. However, an efficient implementation of tensor transpositions is essential to obtain high performance for tensor contractions.

4.2 Integration with COMET Compiler

COMET [23] is a domain-specific compiler for tensor algebra developed using the MLIR framework. It supports code generation for dense and sparse tensor contractions targeted at CPUs, GPUs, FPGAs, and other accelerators. This paper focuses on optimizing dense tensor contractions on CPUs with COMET.

The COMET compiler generates code for dense tensor contractions based on the TTGT approach. COMET implements a progressive lowering pipeline and applies optimizations in multiple dialects of MLIR. Domain-specific optimizations such as reordering multi-operand expressions and optimal index permutation for TTGT are performed at the higher-level Tensor Algebra (TA) dialect. The TA dialect is lowered to the `linalg` dialect where tensor contractions get reformulated as a series of `linalg.copy` (transpose) and `linalg.matmul` (GEMM) operations. Transpose and GEMM operations are optimized and then lowered to lower-level dialects.

The `linalg.matmul` operation is optimized using the `opt-matmul-tiling` pass that applies the tiling strategy from the BLIS framework [29]. With the `opt-matmul-mkernel` pass, it can also optionally replace the innermost GEMM computation after tiling with the BLIS microkernel. These optimizations yield high performance for the GEMM operation.

For the `linalg.copy` operation, COMET implements the `opt-dense-transpose` pass to apply naive tiling and loop permutation optimizations. These optimizations for the transpose kernel are not adequate, and its inefficiency causes significant overhead on tensor contractions, especially in cases when they are memory-bound. We thus replace the `opt-dense-transpose` pass in COMET with our progressive lowering pipeline presented in this paper (figure 3). On integrating our approach with COMET, we generate efficient tensor transpose kernels for TGGT-based tensor contractions by applying sophisticated optimizations such as multi-level tiling, loop reordering, and explicit vectorization.

4.3 Performance of Tensor Contractions

We analyze the performance of the resulting tensor contractions generated with our approach for optimized transpose code generation integrated into COMET. This is compared to the code generated using COMET's `opt-dense-transpose` default pass. For both cases, the `opt-matmul-mkernel` pass is enabled to use the BLIS microkernel for high-performance GEMM kernels. We use a set of 20 tensor contractions from the TCCG benchmark suite [25] for evaluation. Experiments are run on Frontier's AMD CPUs (table 1).

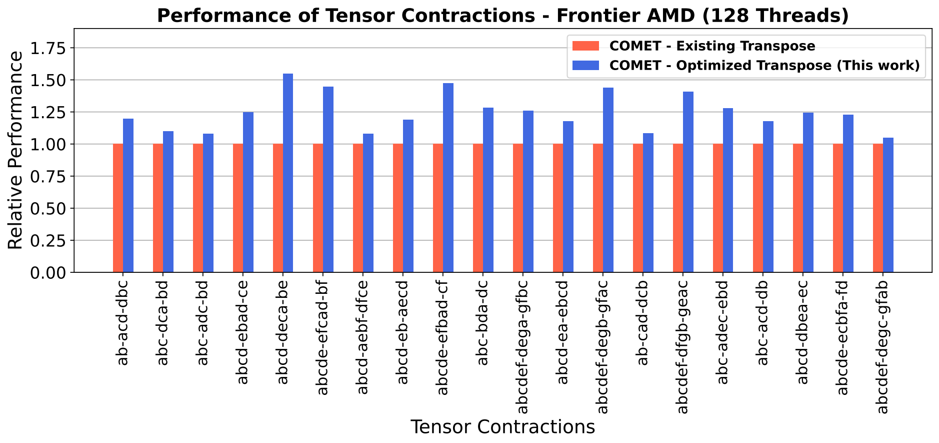


Fig. 7: COMET's performance using our transpose code normalized with COMET's performance using existing transpose code generated.

Figure 7 shows the performance of tensor contractions with transpose kernels generated using our approach, normalized with the performance of tensor contractions generated with COMET's `opt-dense-transpose` pass. Our approach achieves a maximum speedup of 54% and an average speedup of 26% for the 20 tensor contractions. We obtain a speedup of at least 40% for tensor contractions highly limited by memory bandwidth, i.e., the contractions that are exceedingly bound by the performance of the transpose operation, instead of GEMM. Altogether, we observe performance gains for all 20 tensor contractions with COMET, utilizing the optimized tensor transposition kernels generated with our approach.

4.4 Integration With Other Compilers

We demonstrated the performance improvements for tensor contractions by integrating our proposed transpose code generation into COMET. Other MLIR-based domain-specific compilers for deep learning, notably IREE [2] and PlaidML [5], and others [28, 6, 7] could also benefit from optimized tensor transpose kernels. Deep learning workloads involve data layout transformation at various stages of computation. An appropriate example is convolutions, where data layouts are often transformed from, say *NCHWD*, to other suitable layouts. While there have been efforts to reduce data layout transformations in convolutional neural networks [20, 11, 14], they still remain a bottleneck in various cases.

TensorFlow [8] leverages the Eigen library to perform tensor algebra operations. Given that our transpose code generated achieves significant speedups over Eigen (section 3), the XLA compiler can leverage our code generation for high-performance transpositions, using the MLIR backend. The PlaidML compiler built with MLIR can benefit from efficient transpose kernels when transforming computations based on the Batch-Reduce GEMM (BRGEMM) approach [12]. Other use cases for applying efficient transpositions include neighbor aggregation in graph neural networks and attentions in transformers.

5 Conclusion

We present an optimized code generation for arbitrary tensor transpositions by implementing a progressive lowering pipeline in MLIR. We demonstrate performance portability on four different CPU architectures with an efficiency of 74%. We achieve significant speedups over Eigen, and performance on par with HPTT. Integrating this approach with the COMET compiler yields optimized performance for tensor contractions. The proposed transpose code generation can further be integrated with other MLIR-based domain-specific compilers. Implementing additional optimizations (full vectorization of tiles and analytical model for loop reordering) and extension to GPUs are subjects of future work.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources from the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory supported by the Office of Science of the U.S. Department of Energy under Contract DE-AC05-00OR22725. This research also used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. We also thank Stony Brook Research Computing and Cyberinfrastructure and the Institute for Advanced Computational Science at Stony Brook University for access to the

Ookami computing system, supported by the National Science Foundation grant #1927880. The support and resources from the Center for High Performance Computing at the University of Utah are gratefully acknowledged.

References

1. Firefly - A new compiler and runtime for BEAM languages. https://github.com/GetFirefly/firefly
2. IREE Compiler. https://openxla.github.io/iree/, https://github.com/openxla/iree
3. Microsoft Accera Compiler. https://github.com/microsoft/Accera, https://microsoft.github.io/Accera/
4. PennyLane Catalyst Compiler for hybrid quantum-classical programs. https://github.com/PennyLaneAI/catalyst
5. PlaidML Tensor Compiler. https://github.com/plaidml/plaidml, https://plaidml.github.io/plaidml/
6. SHARK High Performance Machine Learning Distribution. https://github.com/nod-ai/SHARK
7. The Torch MLIR Project. https://github.com/llvm/torch-mlir
8. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: TensorFlow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
9. Ben-Nun, T., Ates, B., Calotoiu, A., Hoefler, T.: Bridging Control-Centric and Data-Centric Optimization. In: Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization. CGO 2023 (2023)
10. Gaël Guennebaud and Benoît Jacob and others: Eigen library. http://eigen.tuxfamily.org (2010)
11. Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D., Henry, G., Pabst, H., Heinecke, A.: Anatomy of high-performance deep learning convolutions on SIMD architectures. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 830–841. IEEE (2018)
12. Georganas, E., Banerjee, K., Kalamkar, D., Avancha, S., Venkat, A., Anderson, M., Henry, G., Pabst, H., Heinecke, A.: Harnessing Deep Learning via a Single Building Block. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 222–233 (2020). https://doi.org/10.1109/IPDPS47924.2020.00032
13. Hammond, J.: Automatically tuned libraries for native-dimension tensor transpose and contraction algorithms (2009)
14. Heinecke, A., Henry, G., Hutchinson, M., Pabst, H.: LIBXSMM: accelerating small matrix multiplications by runtime code generation. In: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 981–991. IEEE (2016)
15. Hu, P., Lu, M., Wang, L., Jiang, G.: TPU-MLIR: A Compiler For TPU Using MLIR. arXiv preprint arXiv:2210.15016 (2022)
16. Hynninen, A.P., Lyakh, D.I.: cuTT: A high-performance tensor transpose library for cuda compatible gpus. arXiv preprint arXiv:1705.01598 (2017)
17. Jin, T., Bercea, G.T., Le, T.D., Chen, T., Su, G., Imai, H., Negishi, Y., Leu, A., O'Brien, K., Kawachiya, K., et al.: Compiling ONNX neural network models using MLIR. arXiv preprint arXiv:2008.08272 (2020)
18. Jodra, J.L., Gurrutxaga, I., Muguerza, J.: Efficient 3D transpositions in graphics processing units. International Journal of Parallel Programming **43**(5), 876–891 (2015)

19. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: MLIR: Scaling compiler infrastructure for domain specific computation. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 2–14. IEEE (2021)
20. Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., Wang, Y.: Optimizing {CNN} model inference on {CPUs}. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 1025–1040 (2019)
21. Lyakh, D.I.: An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. Computer Physics Communications **189**, 84–91 (2015)
22. McCalpin, J.D., et al.: Memory bandwidth and machine balance in current high performance computers. IEEE computer society technical committee on computer architecture (TCCA) newsletter **2**(19-25) (1995)
23. Mutlu, E., Tian, R., Ren, B., Krishnamoorthy, S., Gioiosa, R., Pienaar, J., Kestor, G.: COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry. arXiv preprint arXiv:2102.06827 (2021)
24. Pennycook, S.J., Sewall, J.D., Lee, V.W.: Implications of a metric for performance portability. Future Generation Computer Systems **92**, 947–958 (2019)
25. Springer, P., Bientinesi, P.: Design of a high-performance GEMM-like tensor–tensor multiplication. ACM Transactions on Mathematical Software (TOMS) **44**(3), 1–29 (2018)
26. Springer, P., Hammond, J.R., Bientinesi, P.: TTC: A high-performance compiler for tensor transpositions. ACM Transactions on Mathematical Software (TOMS) **44**(2), 1–21 (2017)
27. Springer, P., Su, T., Bientinesi, P.: HPTT: A high-performance tensor transposition C++ library. In: Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming. pp. 56–62 (2017)
28. Tillet, P., Kung, H.T., Cox, D.: Triton: an intermediate language and compiler for tiled neural network computations. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. pp. 10–19 (2019)
29. Van Zee, F.G., Van De Geijn, R.A.: BLIS: A framework for rapidly instantiating BLAS functionality. ACM Transactions on Mathematical Software (TOMS) **41**(3), 1–33 (2015)
30. Vedurada, J., Suresh, A., Rajam, A.S., Kim, J., Hong, C., Panyala, A., Krishnamoorthy, S., Nandivada, V.K., Srivastava, R.K., Sadayappan, P.: TTLG - An efficient tensor transposition library for GPUs. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 578–588. IEEE (2018)
31. Wei, L., Mellor-Crummey, J.: Autotuning tensor transposition. In: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops. pp. 342–351. IEEE (2014)