

# HPTT: A High-Performance Tensor Transposition C++ Library

Paul Springer

AICES, RWTH Aachen University,  
Germany  
springer@aices.rwth-aachen.de

Tong Su

RWTH Aachen University, Germany  
tong.su@rwth-aachen.de

Paolo Bientinesi

AICES, RWTH Aachen University,  
Germany  
pauldj@aices.rwth-aachen.de

## Abstract

Recently we presented TTC, a domain-specific compiler for tensor transpositions. Despite the fact that the performance of the generated code is nearly optimal, due to its offline nature, TTC cannot be utilized in all the application codes in which the tensor sizes and the necessary tensor permutations are determined at runtime. To overcome this limitation, we introduce the open-source C++ library **High-Performance Tensor Transposition (HPTT)**. Similar to TTC, HPTT incorporates optimizations such as blocking, multi-threading, and explicit vectorization; furthermore it decomposes any transposition into multiple loops around a so called micro-kernel. This modular design—inspired by BLIS—makes HPTT easy to port to different architectures, by only replacing the hand-vectorized micro-kernel (e.g., a  $4 \times 4$  transpose). HPTT also offers an optional autotuning framework—guided by performance heuristics—that explores a vast search space of implementations at runtime (similar to FFTW). Across a wide range of different tensor transpositions and architectures (e.g., Intel Ivy Bridge, ARMv7, IBM Power7), HPTT attains a bandwidth comparable to that of SAXPY, and yields remarkable speedups over Eigen’s tensor transposition implementation. Most importantly, the integration of HPTT into the Cyclops Tensor Framework (CTF) improves the overall performance of tensor contractions by up to  $3.1 \times$ .

**Categories and Subject Descriptors** D.1.3 [Concurrent Programming]: Parallel programming; G.4 [Mathematical Software]: Parallel and vector implementations; I.1 [Symbolic and Algebraic Manipulation]: Languages and Systems—Special-purpose algebraic systems

**Keywords** multidimensional transposition, High-Performance Computing, vectorization, tensors, autotuning

## 1. Introduction

Tensors, or multidimensional arrays, are ubiquitous in various scientific fields such as machine learning [1, 24], quantum chemistry calculations [2, 8], multidimensional Fourier transforms [5, 16] and climate simulations [4]. The manipulation of tensors, via operations such as transposition, contraction,<sup>1</sup> completion, and factor-

ization, are performance critical tasks. This work introduces a high-performance library for tensor transpositions.

Transpositions are preparatory tasks that play a role within many other operations. For instance, tensor contractions can be cast in terms of general matrix-matrix multiplications (GEMM), an operation for which there exist numerous highly-optimized implementations (e.g., BLIS, OpenBLAS, MKL); however, the approach is only useful in combination with an efficient and flexible tensor transposition kernel. Despite the unfavourable memory access patterns that arise in high-dimensional transpositions, recent work [12, 20, 21, 26] demonstrated that a domain-specific compiler, such as our Tensor Transposition Compiler (TTC), is capable of generating highly-efficient routines, both for CPUs and accelerators, for a given transposition and problem sizes. While TTC delivers good performance, it is only applicable to transpositions for which the size and the required permutations are known at compile time. To overcome this issue, we designed **High Performance Tensor Transposition (HPTT)**, an open-source C++ library for transpositions of the form

$$\mathcal{B}_{\Pi(i_1 i_2 \dots i_N)} \leftarrow \alpha \times \mathcal{A}_{i_1 i_2 \dots i_N} + \beta \times \mathcal{B}_{\Pi(i_1 i_2 \dots i_N)}, \quad (1)$$

where  $\mathcal{A}$  and  $\mathcal{B}$  are  $N$ -dimensional tensors,  $\Pi(i_1 i_2 \dots i_N)$  denotes an arbitrary permutation of the indices  $i_1, i_2, \dots, i_N$ , and  $\alpha$  and  $\beta$  are scalars. This form enables HPTT to transpose  $\mathcal{A}$  into  $\mathcal{B}$ , and also to scale either of the operands.

Throughout this publication we adopt the Fortran memory layout. Thus, storing the tensor indices from left to right; given an  $N$ -dimensional tensor  $\mathcal{A}_{i_1 i_2 \dots i_N}$ ,  $i_1$  and  $i_N$  respectively are the *fastest-varying* (also known as *stride-1*) and *slowest-varying* indices.

The remainder of this paper is structured as follows. Section 2 outlines related work. The design and structure of HPTT are discussed in Sec. 3, while Sec. 4 contains a performance evaluation, with a breakdown of the used optimization techniques. Conclusions are drawn in Sec. 5.

## 2. Related Work

Two-dimensional tensor transposition (i.e., matrix transposition) is a well studied operation, including optimizations for blocking, vectorization, unrolling, and software prefetching [3, 6, 11, 13, 14, 25]. The same optimizations are investigated in the context three-dimensional out-of-place tensor transpositions on CPUs [10, 22].

The optimization of arbitrary-dimensional tensor transpositions has gained more interest in recent years [12, 20, 21, 26].

Wei et al. [26] presented a code generator which “uses exhaustive global search”, explores blocking, in-cache buffers to avoid conflict misses, loop unrolling, software prefetching and vectorization. While their implementation exhibits good performance on the selected architectures, neither parallelization, nor different loop orders have been considered.

<sup>1</sup> A tensor contraction is the generalization of a matrix-matrix multiplication.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author(s). Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

Array 17 June 18, 2017, Barcelona, Spain  
Copyright © 2017 held by owner/author(s). Publication rights licensed to ACM.  
ACM na. . \$15.00  
DOI: <http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn>

Our previous work on the Tensor Transposition Compiler (TTC) [20, 21] relied on *code generation* to yield a nearly-optimal implementations for any given tensor transposition.

Lyakh et al. [12] designed a generic multidimensional transpose algorithm and evaluated it across different architectures (e.g., Intel Xeon, Intel Xeon Phi, AMD and NVIDIA K20X). Despite the fact that their algorithm outperforms a naive baseline implementation, the results suggest that there still exists a noticeable performance gap to the bandwidth attained by a direct copy.

The recently released CUDA Tensor Transpose (cuTT) library [9] provides high-performance tensor transpositions for NVIDIA GPUs. Despite the fact the HPTT and cuTT target different architectures, both libraries use similar concepts (e.g., an FFTW-like autotuning approach based on plans). Given the different platforms targeted by cuTT and HPTT, these libraries complement each other well.

### 3. High-Performance Tensor Transpositions

HPTT is an open-source<sup>2</sup> high-performance C++ library for out-of-place tensor transpositions running on shared-memory systems. It inherits its key design principle from TTC: any tensor transposition is decomposed into independent 2D tensor transpositions; each of these 2D transpositions is then computed by a so-called *macro-kernel* that is again broken down into loops around a (smaller) *micro-kernel* (see Section 3.2 for details). This design—inspired by BLIS [23]—allows HPTT to be easily ported to different architectures because only the micro-kernel needs be manually implemented via vector intrinsics. Like TTC, HPTT still exhibits desirable properties such as (optional) autotuning (Section 3.1), multi-threading support (Section 3.2), and explicit vectorization (Section 3.2), yielding high performance across a wide range of tensor transpositions and architectures.

The key differences with respect to TTC are the following: First and foremost, HPTT does not require recompilation for different tensor transpositions and sizes; this makes HPTT applicable to scenarios where those parameters can only be determined at runtime (e.g., within CTF [18] or Eigen [7]). Second, HPTT is able to search for different *parallelization strategies* (see Section 3.2) and it avoids the search for different blocking sizes (see Section 3.2). Finally, in contrast to TTC, HPTT needs to perform the autotuning at runtime. To this end, we adopted a recursive design—much like BLIS [23]—which takes an additional parameter that encodes the execution process for any given tensor transposition, henceforth called *plan*.<sup>3</sup>

#### 3.1 Plan-Creation and Autotuning

```

1 // corresponds to i1
2 Plan *i1 = new Plan(0 /*start*/, 6 /*end*/, 1 /*inc*/,
3   1 /*strideA*/, 4 /*strideB*/, NULL);
4 // corresponds to i2
5 Plan *i2 = new Plan(0 /*start*/, 4 /*end*/, 1 /*inc*/,
6   6 /*strideA*/, 1 /*strideB*/, i1);

```

**Figure 1:** Plan data structure example for a single-threaded, unblocked 2D transposition:  $\mathcal{B}_{i_2 i_1} \leftarrow \mathcal{A}_{i_1 i_2}$ ,  $\mathcal{A} \in \mathbb{R}^{6 \times 4}$ ,  $\mathcal{B} \in \mathbb{R}^{4 \times 6}$ .

A plan encodes the execution of any given tensor transposition. As such, a plan represents all loops, each corresponding to a different tensor index. Figure 1 outlines the plan for a single-threaded,

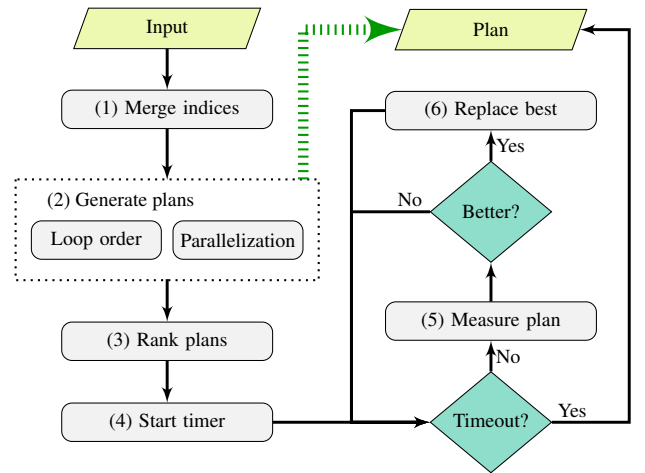
<sup>2</sup>Published under LGPLv3 at [www.github.com/springer13/hptt](http://www.github.com/springer13/hptt).

<sup>3</sup>The plan is conceptually very similar to what is called a *control tree* (cntl\_t) in BLIS.

unblocked 2D transposition  $\mathcal{B}_{i_2 i_1} \leftarrow \mathcal{A}_{i_1 i_2}$ ,  $\mathcal{A} \in \mathbb{R}^{6 \times 4}$ ,  $\mathcal{B} \in \mathbb{R}^{4 \times 6}$ . The plans *i1* and *i2* respectively correspond to indices  $i_1 \in \{0, 1, \dots, 5\}$  and  $i_2 \in \{0, 1, 2, 3\}$ . More precisely, the strides of *i1* with regard to  $\mathcal{A}$  and  $\mathcal{B}$  are 1 and 4, respectively. The increment (*inc*) is not important for now, since this plan corresponds to an unblocked implementation (we consider blocking in Section 3.2).

The loop order is determined by the order *i1* and *i2* point to one another: *i2* stores a pointer to *i1* (Line 6), while *i1*’s pointer is set to NULL (Line 3), indicating that no more indices follow and that the macro-kernel should be invoked. This example illustrates an execution where *i1* and *i2* respectively correspond to the fastest-varying and the slowest-varying index (i.e., the loop associated to  $i_1$  is the innermost). A different loop order can be obtained by having *i1* point to *i2* and *i2* point to NULL. This flexible design enables HPTT to generate different plans—based on different loop orders—quite easily at runtime; likewise, different *parallelization strategies* can be accommodated as well (see Section 3.2).

Figure 2 illustrates the *plan creation* process of HPTT. Given the input—in form of the permutation, size, number of threads and a timeout parameter (for autotuning)—HPTT begins by (1) merging indices which are consecutive in both tensors into a “superindex”.<sup>4</sup> HPTT’s plan creation process offers two different execution paths: The *autotune path* and the *quick path*. The former (2) generates and (3) ranks **all** plans followed by the autotuning process (4)–(6). During the *quick path* (denoted by the green dashed arrow), on the other hand, only a **single**, good plan is created and returned immediately without the need to generate all plans. The *quick path* is essential for keeping the overhead—due to the plan creation—as low as possible when no autotuning is desired (e.g., from within CTF).

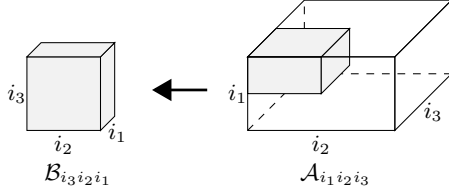


**Figure 2:** HPTT’s plan creation process. The dashed green arrow denotes the *quick path* which returns a good plan immediately.

Whether HPTT takes the *autotune path* or the *quick path* depends on the user-specified timeout parameter; the user, thus, has to decide if the overhead due to the autotuning process can be amortized by repeated executions of the same tensor transposition (e.g., [17])—albeit with (possibly) different data. If the *au-*

<sup>4</sup> For instance,  $\mathcal{B}_{i_2 i_3 i_1} \leftarrow \mathcal{A}_{i_1 i_2 i_3}$  becomes  $\mathcal{B}_{(i_2 i_3) i_1} \leftarrow \mathcal{A}_{i_1 (i_2 i_3)}$ , see [20].

*totune path* is chosen, then (4) a timer is started and HPTT evaluates the performance of the ranked plans (from good to bad) until the timeout has been reached. This autotuning feature is conceptually very similar to that of FFTW [5]. Like FFTW, HPTT also uses the original input data (i.e., the pointers to  $\mathcal{A}$  and  $\mathcal{B}$ ) for autotuning. However, in contrast to FFTW, HPTT does not modify the elements of either input during the autotuning process. We achieve this by setting the coefficient  $\alpha = 0$  and  $\beta = 1$ ; thus, the output tensor is only overwritten with its original value, while  $\mathcal{A}$  is still read from main memory but it is then multiplied with zero.<sup>5</sup>



(a) Visualization.

```

1 // specify permutation and size
2 std::vector<uint32_t> perm = {2,1,0};
3 std::vector<uint32_t> sizeA = { 8, 16, 16};
4 std::vector<uint32_t> outerSizeA = {16, 32, 32};
5 std::vector<uint32_t> outerSizeB = {16, 16, 8};
6
7 // create a plan
8 double timeout = 1.0; // in seconds
9 auto plan = hptt::create_plan(perm,
10 1.0 /*alpha*/, A, sizeA, outerSizeA,
11 0.0 /*beta*/, B, outerSizeB,
12 numThreads, timeout);
13
14 // execute the transposition
15 plan->exec();

```

(b) HPTT input.

**Figure 3:** Exemplary tensor transposition  $\mathcal{B}_{i_3 i_2 i_1} \leftarrow \mathcal{A}_{i_1 i_2 i_3}$  for  $\mathcal{A} \in \mathbb{R}^{16 \times 32 \times 32}$ , the shaded sub-tensor in  $\mathcal{A}$  is of size  $8 \times 16 \times 16$ .

Figure 3 visualizes an exemplary use case of HPTT where a sub-tensor of  $\mathcal{A} \in \mathbb{R}^{16 \times 32 \times 32}$  is transposed-and-compacted into  $\mathcal{B} \in \mathbb{R}^{16 \times 16 \times 8}$  via the tensor transposition  $\mathcal{B}_{i_3 i_2 i_1} \leftarrow \mathcal{A}_{i_1 i_2 i_3}$ . HPTT accepts so called *outer sizes* (similar to the leading dimension in BLAS) which enable HPTT to operate on sub-tensors; this feature makes it possible to transpose-and-compact (see Figure 3a) or to transpose-and-scatter (reverse operation of Figure 3a, not shown).

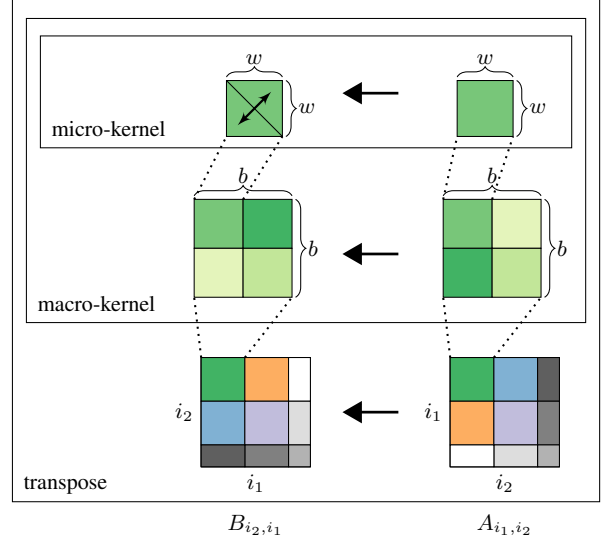
### 3.2 Vectorization and Parallelization

HPTT decomposes an arbitrary-dimensional tensor transposition into many independent two-dimensional  $b \times b$  *macro-tiles* which are composed of  $w \times w$  *micro-tiles* (see Fig. 4). These two-dimensional tiles are, always chosen such that the stride-1 index in both  $\mathcal{A}$  and  $\mathcal{B}$  is preserved, facilitating fully-vectorized memory operations and thereby exploiting the spatial locality inherent to tensor transpositions. As illustrated in Fig. 4, the macro- and micro-tiles are respectively computed by a *macro-* and *micro-kernel*.

Each  $w \times w$  micro-tile denote an explicitly vectorized, in-register transposition, with  $w$  corresponding to the vector-width of the underlying architecture (e.g.,  $w = 8$  for single-precision elements on an AVX-enabled processor). The interested reader is referred to [20] for further details on the vectorization.

A noticeable difference to TTC [20, 21], however, is that the remainder (the macro-tiles shaded in gray, Fig. 4) are now also vectorized. HPTT starts out with  $b = 4w$ , once the remainder is

<sup>5</sup>  $\mathcal{A}$  may not have special values like  $\text{inf}$  or  $\text{NaN}$ .



**Figure 4:** Decomposition of a 2D transposition into  $b \times b$  macro-tiles and  $w \times w$  micro-tiles.

reached  $b$  is decreased to  $b/2$ ; this concept continues until  $b = w$ . Thanks to this concept, HPTT does not search for suitable blocking parameters any longer and removes this search direction entirely, making the plan generation process less complex.

The macro-tiles are completely independent from one another and can be computed in-parallel by different threads. HPTT is able to parallelize the loops, corresponding to the tensor indices, individually. For instance, revisiting Fig. 1, two threads  $t_1$  and  $t_2$  could parallelize the loop associated to  $i_1$ , if  $t_1$  uses  $i1 \rightarrow \text{start}=0$ ;  $i1 \rightarrow \text{end}=3$ ; while  $t_2$  uses  $i1 \rightarrow \text{start}=3$ ;  $i1 \rightarrow \text{end}=6$ ; the loop corresponding to  $i_2$  could also be parallelized similarly. Likewise, HPTT can parallelize multiple loops simultaneously; this mechanism enables a new search direction which had not been present in TTC before.

### 3.3 Performance Heuristics

Given the vast search space of viable plans, the challenge is to select a good plan that yields high performance. This section outlines the performance heuristics of HPTT which addresses this problem. The performance model consists of two separate heuristics: the parallelization heuristic (see Section 3.3.1) and the loop order heuristic (see Section 3.3.2).

#### 3.3.1 Parallelization

As mentioned above, HPTT is able to parallelize all loops individually resulting in a large search space of different parallelization strategies.

The fundamental ideas behind the parallelization heuristic are manifold: First and foremost, load-balancing should be maximized; thus, total amount of work should be equally distributed among the threads. Second, the parallelization of any stride-1 index should be avoided; abiding to this rule increases the amount of consecutive memory accesses per thread. Finally, if one of the stride-1 indices needs to be parallelized in order to increase load-balancing, then this heuristic prefers to parallelize the stride-1 index of  $\mathcal{A}$  over the stride-1 index of the output tensor  $\mathcal{B}$ ; the rationale being that *false sharing* between the threads should be avoided. An analytical description of this heuristic is beyond the scope of this paper;

however, a curious reader can find the corresponding source code at [www.github.com/springer13/hptt](http://www.github.com/springer13/hptt).

### 3.3.2 Loop Order

A tensor transposition of  $N$ -dimensional tensors has  $N!$  distinct ways to order the loops. Most of these loop orders lead to significantly different performance [20], making it critical to choose a good loop order for any given tensor transposition.

We encode a loop order  $L$  of an  $N$ -dimensional tensor transposition as an  $N$ -tuple  $L = (l_1, l_2, \dots, l_N)$  with  $l_i \in \{1, 2, \dots, N\}$  such that the loop corresponding to index  $l_i$  represents the  $i$ -th loop around the macro-kernel. For instance, given the tensor transposition  $\mathcal{B}_{i_6 i_5 i_4 i_1 i_3 i_2} \leftarrow \mathcal{A}_{i_1 i_2 i_3 i_4 i_5 i_6}$  and the loop order  $L = (6, 5, 4, 1, 3, 2)$  means that any *plan* that uses  $L$  traverses  $\mathcal{B}$  in a linear fashion. While such a loop order is ideal for  $\mathcal{B}$ , it would (most likely) be suboptimal with respect to the memory accesses to  $\mathcal{A}$ . More precisely, the loop corresponding to the stride-1 index of  $\mathcal{A}$  ( $i_1$ ) is the fourth loop around the macro-kernel, leading to a strided memory access pattern with a large stride.

The rationale behind HPTT’s loop heuristic is that loops corresponding to innermost indices of either of  $\mathcal{A}$  and  $\mathcal{B}$  should be “close” to the macro-kernel. Moreover, this heuristic slightly favours the innermost indices of  $\mathcal{B}$  over those of  $\mathcal{A}$  to favour consecutive writes over consecutive reads.

Rank	Permutation		
	6,5,4,3,2,1	4,3,6,2,1,5	6,1,5,4,3,2
1	6,1,5,2,4,3	4,1,3,2,6,5	1,6,2,5,3,4
2	6,1,2,5,4,3	1,4,3,2,6,5	1,6,2,3,5,4
3	1,6,5,2,4,3	4,1,2,3,6,5	1,6,5,2,3,4
	(a)	(b)	(c)

**Table 1:** Top-3 loop orders ranked by the loop heuristic for three different permutations. The leftmost index denotes the innermost loop, while the rightmost index corresponds to the outermost loop.

Table 1 shows the top-3 loop orders for three different tensor transpositions (a) – (c). We observe that the stride-1 indices are always either the fastest-varying (innermost) or the second fastest-varying index. For instance, the top-1 loop order for the tensor transposition  $\mathcal{B}_{i_6 i_5 i_4 i_1 i_3 i_2} \leftarrow \mathcal{A}_{i_1 i_2 i_3 i_4 i_5 i_6}$  in column (a) interleaves the indices of  $\mathcal{B}$  and  $\mathcal{A}$  while given precedence to  $\mathcal{B}$ . On other hand, the top-1 loop order for the tensor transposition  $\mathcal{B}_{i_6 i_1 i_5 i_4 i_3 i_2} \leftarrow \mathcal{A}_{i_1 i_2 i_3 i_4 i_5 i_6}$  in column (c) chooses  $i_1$  as the innermost index due to the fact that  $i_1$  is the second fastest-varying index in  $\mathcal{B}$  while  $i_6$  (the fastest-varying index of  $\mathcal{B}$ ) is the slowest-varying index (i.e., least important) index of  $\mathcal{A}$ ; phrased differently,  $i_1$  is important to both  $\mathcal{A}$  and  $\mathcal{B}$ , while  $i_6$  is only important to  $\mathcal{B}$ .

### 3.4 Recursive Design

Figure 5 illustrates the recursive design of HPTT. Each invocation of `transpose()` either invokes the macro-kernel (Lines 7-9) or enters a loop and then unfolds the plan (Lines 12-14) with a recursive call. In the latter case, the pointers to  $\mathcal{A}$  and  $\mathcal{B}$  are offset according to the strides of the respective tensor index and the plan is unfolded (Line 14) before making the recursive.

## 4. Performance Evaluation

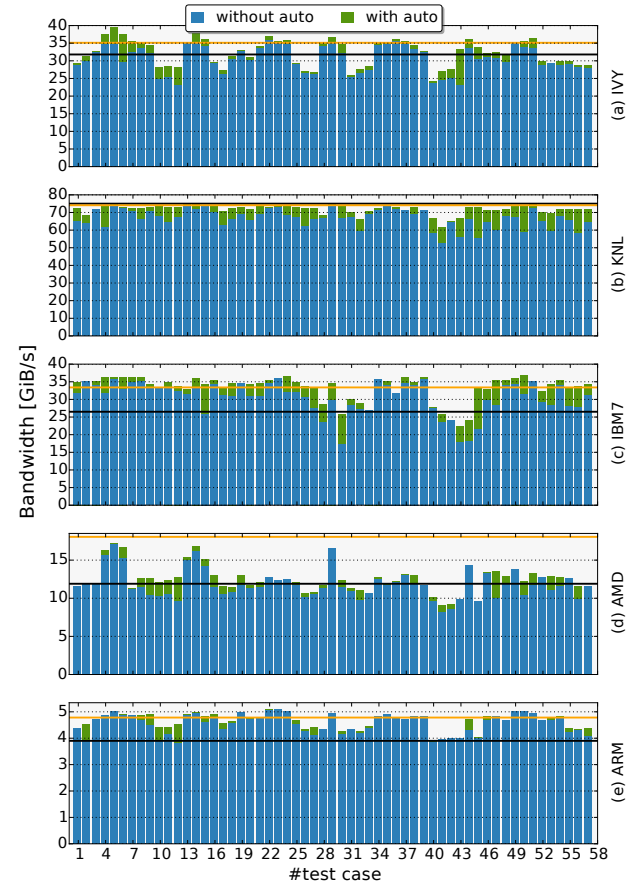
To assess the performance of HPTT across a wide range of use-cases, we report the bandwidth attained on a tensor transpositions benchmark [20] that comprises a total of 57 transpositions ranging from 2D to 6D, with each tensor roughly occupying 200 MB of memory. All measurements are based on the maximum bandwidth attained over multiple executions (caches are cleared in between runs). If not otherwise noted, we use single-precision tensors, initialized in a NUMA-friendly fashion to distribute them

```

1 void transpose( const float* A, float alpha,
2                float* B, float beta, const Plan* plan){
3     int strideA = plan->strideA;
4     int strideB = plan->strideB;
5     if( isMacroKernel(plan) ){
6         /** invoke macro-kernel */
7         for(int i = plan->start; i < end; i+= plan->inc)
8             macroKernel( &A[i*strideA], alpha, plan->next->strideA,
9                          &B[i*strideB], beta, plan->next->strideB);
10    } else {
11        /** recurse */
12        for(int i = plan->start; i < plan->end; i+= plan->inc)
13            transpose( &A[i*strideA], alpha,
14                      &B[i*strideB], beta, plan->next);
15    }
16 }

```

**Figure 5:** Plan Execution. This function has been simplified from its original form for better readability.



**Figure 6:** HPTT bandwidth with and without autotuning. Horizontal orange and black lines respectively denote the system’s SAXPY and STREAM bandwidth.

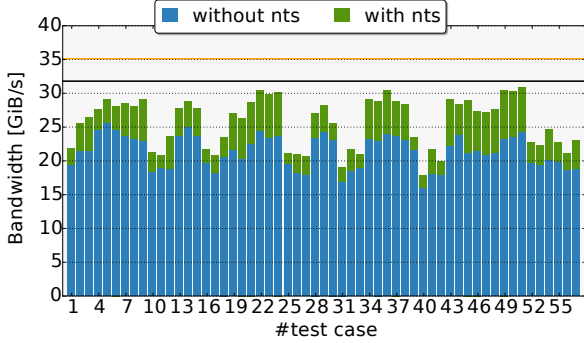
evenly among the memory controllers. The bandwidth is calculated as

$$\text{Bandwidth} := \frac{\lambda \times S}{1024^3 \times \text{Time}} \text{ [GiB/s]}, \quad (2)$$

where  $S$  denotes the size of the transposed tensor (in bytes) and  $\lambda$  is either 2 or 3, depending on whether  $\mathcal{B}$  is overwritten ( $\beta = 0$ ) or updated ( $\beta \neq 0$ ); unless otherwise stated, we report the results for  $\beta \neq 0$ .

Name	Microarchitecture	Model	#Cores	#Threads	Bandwidth [GiB/s]		Compiler	Compiler-flags
					SAXPY	STREAM		
IVY	Ivy Bridge	Intel E5-2670 v2	$2 \times 10$	$2 \times 10$	35.1	31.8	icpc 16.0.3	-O3 -xHost
KNL	Knights Landing	Intel Xeon Phi 7210	64	64	74.1	75.0	icpc 17.0.2	-O3 -xHost
IBM	Power7	IBM PowerPC A2	16	64	33.4	26.5	g++ 6.3	-O3 -mcpu=native
STR	AMD Steamroller	A10-7850K	4	4	18.1	11.9	g++ 5.3	-O3 -march=native
ARM	ARMv7-A	ODROID-XU3	$4 + 4$	8	4.8	3.9	g++ 5.4	-O3 -march=native

**Table 2:** Hardware description. The *SAXPY* and *STREAM* columns indicate the bandwidth attained for  $\mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y}$ , and for the  $\mathbf{z} \leftarrow \alpha \mathbf{x} + \mathbf{y}$  (*STREAM triad* [15]), respectively.



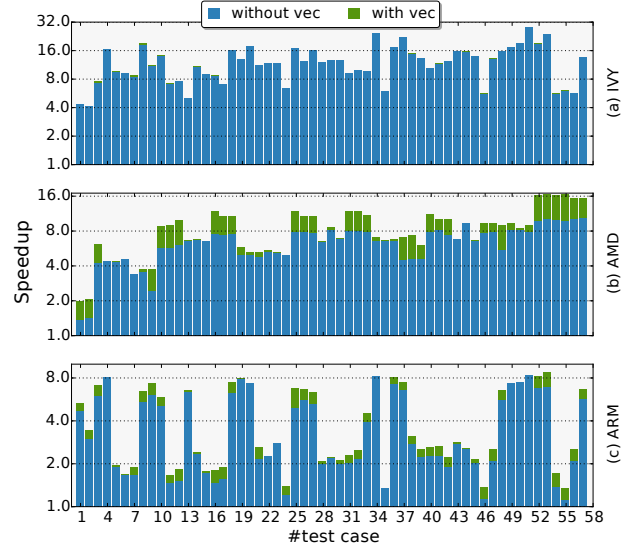
**Figure 7:** HPTT bandwidth for  $\beta = 0$  on Ivy Bridge system. Horizontal orange and black lines denote the system’s SAXPY and STREAM bandwidth, respectively.

We evaluate HPTT’s performance on five platforms (Table 2); the number of threads, per platform, were chosen empirically to give the best performance.

In Fig. 6, we observe that (1) on average, HPTT attains the following fraction of the SAXPY / STREAM bandwidth for the various systems: 92% / 102% (IVY), 97% / 96% (KNL), 100% / 126% (IBM), 69% / 105% (AMD) and 97% / 119% (ARM); these results illustrate HPTT’s close-to-optimal performance. (2) Autotuning, on average, only improves the performance by 6%, 8%, 10%, 6%, and 2% for the IVY, KNL, IBM, AMD and ARM system, respectively; these numbers indicate that the performance heuristics work well in most cases. (3) The maximum speedup due to autotuning can be as high as  $1.45\times$ ,  $1.31\times$ ,  $1.53\times$ ,  $1.35\times$  and  $1.19\times$  for the IVY, KNL, IBM, AMD and ARM system, respectively; thus, making autotuning a viable option for those cases where the autotuning overhead can be amortized over several tensor transpositions.

Figure 7 covers the case  $\beta = 0$  (i.e., an out-of-place transposition without accumulating into the output) on the IVY system, and highlights the impact of non-temporal stores. While the overall performance is lower than what outlined in Fig. 6 ( $\beta \neq 0$ ), on average non-temporal stores improve the performance by  $1.20\times$ .

HPTT’s speedup over Eigen [7], with and without explicit vectorization, are shown in Fig. 8. The maximum speedup for the IVY, AMD, and ARM systems is as high as  $27.4\times$ ,  $16.6\times$ , and  $8.8\times$ , respectively. Explicit vectorization does not affect the performance on the Ivy Bridge (or and KNL systems; data not shown); we take this as a clear indication of icpc’s superb optimization capabilities. Vice versa, both the AMD and ARM systems experience noticeable average speedups of  $1.28\times$  and  $1.12\times$  due to explicit vectorization, suggesting that gcc struggles to find a good vectorization.



**Figure 8:** HPTT speedup over Eigen with and without explicit vectorization using a semi-log scale.

#### 4.1 Cyclops Tensor Framework

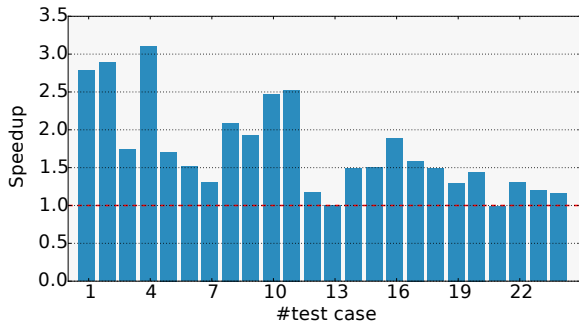
We analyze now the effects of HPTT on CTF [18].<sup>6</sup> Since one tensor contraction within CTF can require up to six tensor transpositions, highly-optimized tensor transpositions are critical to achieve good overall performance. We measure the speedup over CTF for a tensor contraction  $\chi$  as

$$\text{Speedup}(\chi) = \frac{\text{Time}(\text{CTF without HPTT}(\chi))}{\text{Time}(\text{CTF with HPTT}(\chi))}. \quad (3)$$

Figure 9 shows that HPTT improves CTF’s performance noticeably; the test cases are sorted identically to those presented in [19], that is, bandwidth-bound and compute-bound contractions on the left and right, respectively. As expected, we observe larger speedups (up to  $3.1\times$ ) for those contractions limited by memory bandwidth, than for the compute-bound ones. This is intuitive because the contractions towards the left spend more time on transpositions than those towards the right, which are instead dominated by a matrix-matrix multiplication (GEMM); test cases 13 and 19 are indeed pure GEMMs and thus do not require any transposition. Finally, we stress that these timings also include the time to create the “best” plan (according to HPTT’s performance heuristics); the plan creation overhead is—thanks to the *quick path* (see Section 3.1)—negligible (on average, less than 0.1%).

<sup>6</sup> While CTF targets distributed-memory systems, it uses a shared-memory tensor transposition (via OpenMP) per MPI rank.





**Figure 9:** HPTT's impact on CTF's performance using double precision on the IVY system.

## 5. Conclusions and Future Directions

We introduced HPTT, an open-source<sup>7</sup> C++ library for high-performance tensor transpositions that avoids the code generation process of its predecessor, TTC [20]. At the same time, HPTT still preserves desirable properties such as autotuning, explicit vectorization, blocking, and multi-threading.

HPTT's autotuning framework is able to generate multiple plans; this feature enables HPTT to tune for an optimal plan with respect to any given tensor transposition and size at runtime. We further outlined HPTT's performance heuristics which yield competitive performance to the best plan which was found by the autotuning framework.

HPTT's close-to-optimal performance was demonstrated on a wide range of architectures, suggesting that there is essentially no more performance to be gained. We also assessed the performance of Eigen's tensor transposition implementation, indicated that HPTT executes up to 27.4× faster. Given Eigen's importance for TensorFlow [1], HPTT could also prove to be valuable within the machine learning community.

Provided HPTT's close-to-optimal performance and its rich feature set (e.g., scaling, support for subtensors), this publication concludes our work on tensor transpositions. In the future it would be interesting to see if HPTT's autotuning process can also be applied to the BLIS framework.

Finally, we integrated HPTT into our local copy of CTF, resulting in an average speedup of 1.9× across a wide range of tensor contractions.

## Acknowledgments

Financial support from the Deutsche Forschungsgemeinschaft (DFG) through grant GSC 111 is gratefully acknowledged. Furthermore, we thank Edgar Solomonik for assisting us to integrate HPTT into CTF. We also greatly appreciate the valuable discussions with Field G. Van Zee regarding BLIS' internal design. We also thank Elmar Peise for his helpful feedback to this paper.

## References

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems. 2015. URL <https://www.tensorflow.org>.
- [2] R. J. Bartlett and M. Musiał. Coupled-cluster theory in quantum chemistry. *Reviews in Modern Physics*, 79(1):291–352, 2007.
- [3] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. pages 195–205, 2000. doi: 10.1109/HPCA.2000.824350.

- [4] J. Drake, I. Foster, J. Michalakes, B. Toonen, and P. Worley. Design and performance of a scalable parallel community climate model. *Parallel Computing*, 21(10):1571–1591, 1995.
- [5] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840301.
- [6] G. C. Golubogin. PRIM: A fast matrix transpose method. *IEEE Trans. Software Eng.*, 7(2):255–257, 1981.
- [7] G. Guennebaud, B. Jacob, et al. Eigen v3. <http://eigen.tuxfamily.org>, 2010.
- [8] R. J. Harrison, G. Beylkin, F. A. Bischoff, J. A. Calvin, G. I. Fann, J. Fosso-Tande, D. Galindo, J. R. Hammond, R. Hartman-Baker, J. C. Hill, J. Jia, J. S. Kottmann, M. Y. Ou, L. E. Ratcliff, M. G. Reuter, A. C. Richie-Halford, N. A. Romero, H. Sekino, W. A. Shelton, B. E. Sundahl, W. S. Thornton, E. F. Valeev, Á. Vázquez-Mayagoitia, N. Vence, and Y. Yokoi. MADNESS: A multiresolution, adaptive numerical environment for scientific simulation. *CoRR*, abs/1507.01888, 2015. URL <http://arxiv.org/abs/1507.01888>.
- [9] A. Hynninen and D. I. Lyakh. cuTT: A High-Performance Tensor Transpose Library for CUDA Compatible GPUs. *CoRR*, abs/1705.01598, 2017. URL <https://arxiv.org/abs/1705.01598>.
- [10] J. L. Jodra, I. Gurrutxaga, and J. Muguerza. Efficient 3D transpositions in graphics processing units. *International Journal of Parallel Programming*, pages 1–16, 2015.
- [11] Q. Lu, S. Krishnamoorthy, and P. Sadayappan. Combining analytical and empirical approaches in tuning matrix transposition. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 233–242. ACM, 2006.
- [12] D. I. Lyakh. An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. *Computer Physics Communications*, 189:84–91, 2015.
- [13] G. Mateescu, G. H. Bauer, and R. A. Fiedler. Optimizing matrix transposes using a POWER7 cache model and explicit prefetching. *ACM SIGMETRICS Performance Evaluation Review*, 40(2):68–73, 2012.
- [14] J. McCalpin and M. Smotherman. Automatic benchmark generation for cache optimization of matrix operations. In *Proceedings of the 33rd annual on Southeast regional conference*, pages 195–204. ACM, 1995.
- [15] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.
- [16] D. Pekurovsky. P3DFFT: A framework for parallel computations of Fourier transforms in three dimensions. *SIAM Journal on Scientific Computing*, 34(4):C192–C209, 2012.
- [17] K. Raghavachari, G. W. Trucks, J. A. Pople, and M. Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters*, 157(6):479–483, 1989.
- [18] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 813–824, May 2013. doi: 10.1109/IPDPS.2013.112.
- [19] P. Springer and P. Bientinesi. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. *CoRR*, 2016. URL <http://arxiv.org/abs/1607.00145>.
- [20] P. Springer, J. R. Hammond, and P. Bientinesi. TTC: A high-performance compiler for tensor transpositions. *CoRR*, 2016. URL <http://arxiv.org/abs/1603.02297>.
- [21] P. Springer, A. Sankaran, and P. Bientinesi. TTC: A Tensor Transposition Compiler for Multiple Architectures. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 41–46, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4384-8.

<sup>7</sup> Available at [www.github.com/springer13/hptt](http://www.github.com/springer13/hptt).

- doi: 10.1145/2935323.2935328. URL <http://doi.acm.org/10.1145/2935323.2935328>.
- [22] M. van Heel. A fast algorithm for transposing large multidimensional image data sets. *Ultramicroscopy*, 38(1):75–83, 1991.
- [23] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, 2015. URL <http://doi.acm.org/10.1145/2764454>.
- [24] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation, 2014. URL <http://arxiv.org/abs/1412.7580>.
- [25] A. Vladimirov. Multithreaded transposition of square matrices with common code for Intel Xeon processors and Intel Xeon Phi coprocessors, 2013. URL [https://www.researchgate.net/profile/Andrey\\_Vladimirov/publication/258048110\\_Multithreaded\\_Transposition\\_of\\_Square\\_Matrices\\_with\\_Common\\_Code\\_for\\_Intel\\_Xeon\\_Processors\\_and\\_Intel\\_Xeon\\_Phi\\_Coprocessors/links/00463526c2fa40a1f3000000.pdf](https://www.researchgate.net/profile/Andrey_Vladimirov/publication/258048110_Multithreaded_Transposition_of_Square_Matrices_with_Common_Code_for_Intel_Xeon_Processors_and_Intel_Xeon_Phi_Coprocessors/links/00463526c2fa40a1f3000000.pdf).
- [26] L. Wei and J. Mellor-Crummey. Autotuning tensor transposition. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), IEEE International*, pages 342–351. IEEE, 2014.