TTLG - An Efficient Tensor Transposition Library for GPUs

Jyothi Vedurada*, Arjun Suresh†, Aravind Sukumaran Rajam†, Jinsung Kim†, Changwan Hong†, Ajay Panyala‡, Sriram Krishnamoorthy‡, V. Krishna Nandivada*, Rohit Kumar Srivastava†, P. Sadayappan†
*IIT Madras {vjyothi,nvk}@cse.iitm.ac.in ‡PNNL {ajay.panyala,sriram}@pnnl.gov
†OSU {suresh.86, sukumaranrajam.1, kim.4232, hong.589, srivastava.141, sadayappan.1}@osu.edu

***Abstract*—This paper presents a Tensor Transposition Library for GPUs (TTLG). A distinguishing feature of TTLG is that it also includes a performance prediction model, which can be used by higher level optimizers that use tensor transposition. For example, tensor contractions are often implemented by using the TTGT (Transpose-Transpose-GEMM-Transpose) approach – transpose input tensors to a suitable layout and then use high-performance matrix multiplication followed by transposition of the result. The performance model is also used internally by TTLG for choosing among alternative kernels and/or slicing/blocking parameters for the transposition. TTLG is compared with current state-of-the-art alternatives for GPUs. Comparable or better transposition times for the "repeated-use" scenario and considerably better "single-use" performance are observed.**

I. Introduction

Tensor transposition is an important layout transformation primitive for many domains like machine learning and computational chemistry that use tensors as a core data structure. It involves a permutation of the indices of an input tensor:

$$B_{\rho(i_0,i_1,i_2,\cdots,i_{d-1})} \leftarrow A_{i_0,i_1,i_2,\cdots,i_{d-1}}$$

where A and B are the input and output tensors, respectively, and ρ denotes the index permutation for the transposition.

An arbitrary tensor transposition (of a d-dimensional tensor) can be simply achieved by a d-nested loop that reads each element from the input tensor and writes it to the output tensor element corresponding to the permuted indexing. However, this is often very inefficient because of the high access stride for at least one of the two tensors. Consider the 2D case of transposing a matrix. This can be achieved by a doubly nested loop: `for (i,j) B[i][j] = A[j][i]` or `for (i,j) B[j][i] = A[i][j]`. With either form, one of the two arrays is accessed with a high stride, causing poor data locality (uncoalesced accesses on GPUs) and low performance.

In this paper, we present the design, implementation, and experimental evaluation of TTLG, a Tensor Transposition Library for GPUs. TTLG has the following features:

- It includes a taxonomy of tensor transposition schemas with different data movement strategies, and a model-driven approach to choose the appropriate kernel based on the specified permutation and the extents of the tensor dimensions.
- It provides a performance modeling interface that can be queried by an invoking context (e.g., a higher level library using tensor transposition as a building block) to estimate the time for any tensor transposition.
- It is demonstrated to be comparable or faster than other GPU tensor transposition libraries/code-generators across a range of tensor sizes and permutations.

II. Background and Related Work

Background on GPUs: The hardware in a GPU (we use Nvidia terminology) is organized as a set of streaming multiprocessors (SM), each of which consists of a set of SIMD streaming processors (SP). All SPs in an SM share resources such as registers and shared memory. Shared memory is divided into multiple banks to facilitate simultaneous accesses and thus achieve higher memory bandwidth. Shared memory provides efficient access even for non-contiguous data, as long as there are no bank conflicts. However, if multiple threads access the same memory bank (bank conflict), the accesses are serialized. Array padding can help reduce/avoid bank conflicts.

In a GPU, a single thread represents the finest granularity of execution. Threads are grouped into warps, and all threads in a warp are executed in a lock-step manner in an SM. Warps are grouped to form thread blocks. A grid consists of a set of thread blocks scheduled to be executed on a GPU. We use the terms *threadid*, *warpid* and *blockid* to identify a thread, warp, and thread block, respectively. The warp size on modern GPUs is 32. From here on, we refer to warp size as $\mathcal{WS}$.

Out-of-place Tensor Transposition: Lyakh et al. [1] proposed a generic parallel tensor transposition algorithm for multicore CPUs, Intel Xeon Phi, and NVIDIA Tesla GPUs. Although the proposed algorithm showed a 2–3x performance over a naive algorithm, it was suboptimal in terms of the bandwidths achieved. Hynninen and Lyakh [2] presented cuTT (CUDA Tensor Transpose), an optimized tensor transposition algorithm for NVIDIA GPUs that efficiently uses shared memory and computes the indices (memory positions of tensor elements) in parallel. TTC (Tensor Transposition Compiler) [3] auto-generates C++/CUDA code that exhibits high performance on Intel Haswell, AMD Steamroller, Intel Knights Corner, NVIDIA Kepler, and NVIDIA Maxwell architectures. While efficient, TTC generates optimized code for a specific tensor size and permutation. HPTT [4] (High-Performance Tensor Transposition) overcomes this limitation but does not currently support GPUs. In this paper, we describe TTLG, an efficient tensor transposition library for NVIDIA GPUs. We present an experimental evaluation comparing performance with cuTT and TTC.

Autotuning Tensor Transposition: Lu et al. [5] optimized matrix transposition by combining static analysis and empirical search to tune optimization parameters related to tiling,

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00067

vectorization, memory alignment, etc. Wei et al. [6] presented a three-phase autotuner to improve tensor transposition on a dual-socket system with Intel Westmere processors and an IBM POWER 755 with four Power7 processors. TTC [3] explores candidate implementations that cover a range of combinations of loop orders and blocking to choose the fastest candidate based on measured execution time. cuTT [2] supports two modes to select the tensor transposition algorithm and associated parameters at runtime. In heuristic mode, cuTT employs the MWP-CWP GPU performance model [7], based on the amount of warp-level parallelism. In measurement mode, cuTT creates and executes a number of plans to choose the best configuration. We use a performance model, based on linear regression coefficients developed off-line, to choose the best kernel and configuration parameters. We present an experimental evaluation, comparing performance against TTC and cuTT, with and without including the model/plan time.

Lower Dimensional (2D or 3D) Tensor Transposition: Prior work has focused on improving 2D tensor (matrix) transpositions both for CPUs [5], [8] and GPUs [9]. Jodra et al. [10] proposed algorithms for three-dimensional tensor transposition on GPUs based on simple extensions to 2D tensor transposition algorithms to achieve high performance through coalesced memory accesses. We develop a generic tensor transposition algorithm that can efficiently handle arbitrary dimensional tensor transpositions.

Asymptotic Analysis: Aggarwal et al. [11] present lower bounds on data movement for matrix transposition, under the constraint that efficient data movement requires contiguous blocks of a minimal size B to be moved to/from secondary storage. Their lower bounds expressions imply that when B^2 is larger than the capacity of *fast* memory, multiple passes of data movement between slow and fast memory will be required. Later, we discuss the data-movement complexity of our tensor transposition schemes in terms of the number of cache-line transactions. For the GPU transposition context, B^2 is much smaller than the capacity of L1/L2 cache and therefore corresponds to a single pass for the lower bounds analysis of Aggarwal et al. [11].

III. Overview of Approach

In this section, we use examples and provide an overview of the TTLG approach to tensor transposition. The tensor transposition operation is fully parallel, with no data dependencies. Each elementary data movement can be performed completely independently of any other. However, efficient data movement is the challenge. We first use matrix transposition to highlight how a key requirement for efficient transposition can be achieved on GPUs: coalesced data access. Figure 1 shows the transposition of a 2D 128×128 matrix using 32×32 slices, with each slice being handled by a different thread block. A shared memory buffer of size 32×33 (the extra column is padded to avoid share-memory bank conflicts) is used. Each warp in a thread block copies 4 row-segments in the 2D slice from global-memory to shared-memory, with adjacent threads in the warp moving contiguous elements in the row-segment (row-major linearized layout in memory), thereby ensuring coalesced global-memory access. Different warps bring different row-segments of the input matrix and place them in corresponding locations in shared memory.

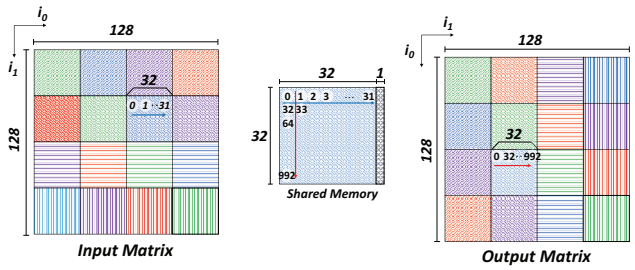


Fig. 1: Matrix transposition

The set of warps then read the buffered data from shared-memory, column-segment by column-segment, and perform coalesced stores into contiguous elements in a row of the result matrix in global memory. Thus efficient coalesced data movement is achieved both for copy-in from global-memory to shared-memory, as well as write-out from shared-memory to global-memory. Without padding, all elements in a column of a 32×32 2D shared-memory buffer would be mapped to the same shared-memory bank (there are 32 banks), causing severe slowdown due to serialization of accesses from a single bank. But the padded 32×33 shared-memory buffer causes staggering of the element-to-bank mapping so that each element in a column of data in the 2D buffer is mapped to a distinct shared-memory bank, preventing any bank conflicts.

The above strategy can be directly applied to higher dimensional tensors. For example, let us consider the tensor transposition $[a, b, c, d] \Rightarrow [d, c, b, a]$, where '$a$' is the fastest varying dimension in the input and 'd' is the fastest varying dimension in the output[1]. Similar to matrix transposition, a warp can read contiguous elements from the input tensor (corresponding to dimension 'a') to shared memory and then write out to contiguous memory locations in the output tensor in coalesced manner. With tensor transposition, a set of contiguous indices can be logically treated as a single 'combined' index.

For example, let the extent of dimensions a, b, c, d be $16, 2, 32, 32$, respectively. If the above matrix transpose approach is followed, only 16 contiguous elements can be read along dimension 'a', and only half of the 32 threads in a warp can be active, leading to underutilization of resources. Instead, 'a' and 'b' can be viewed as a "combined" dimension in the input tensor, since they are laid out contiguously in the tensor. Thus contiguous groups of 32 elements can now be read in from the input tensor to shared memory. In other words, the columns in shared memory are mapped to $a \times b$. Since 'd' is the fastest varying dimension of the output tensor, it is mapped to rows in the 2D shared-memory buffer. While writing out from shared memory, all the threads in a warp will simultaneously read a column of shared memory and write to contiguous locations in the output tensor.

Figure 2 illustrates the transposition approach when the (combined) fastest varying index-sets of the input and output tensor are distinct and sufficiently large (greater than 32). This is one of four transposition schemas incorporated in TTLG,

[1]While TTLG is implemented in C and uses a row-major linearized view of multidimensional arrays, we use the MATLAB/Fortran convention for the abstract notation to stay consistent with previous publications on this topic.

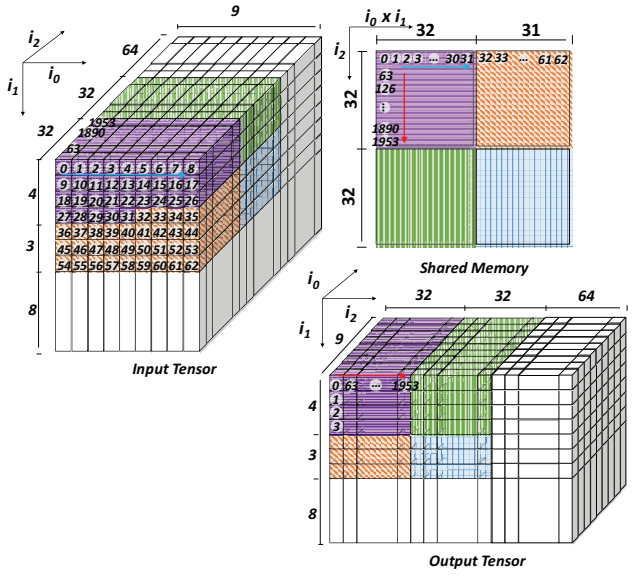


Fig. 2: Tensor transposition (0, 1, 2) → (2, 1, 0): Orthogonal-Distinct

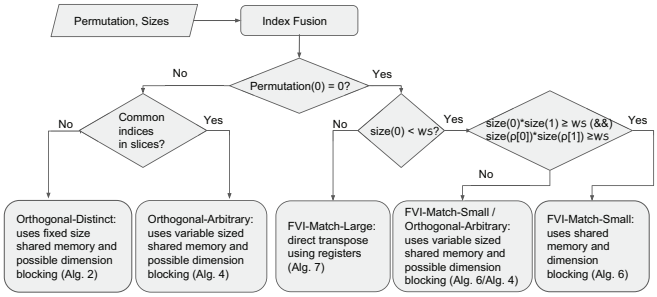


Fig. 3: Taxonomy of schema for tensor transposition

Algorithm 1: Taxonomy

Input : $in\{i_0, i_1, \cdots, i_N\}$; B: Required slice size [Alg. 3]
ρ: Permutation function, maps indices from input to output
Output: $out\{\rho(i_0, i_1, \cdots, i_N)\}$

```
function taxonomy()
    I = ∅; I_vol = 1
    index = 0 // fvi
    while (I_vol < B) do
        I_vol = I_vol * dim(index)
        I = I ∪ {index}
        index = index + 1
    // similarly compute O and O_vol
    if (I ∩ O == ∅) then
        transposeOrthogonalDistinct()
    else if (i_0 == ρ(i_0)) then
        if (dim(i_0) ≥ WS) then
            transposeFVIndicesMatchLarge()
        else if ((dim(i_0) * dim(i_1) ≥ WS) AND (dim(ρ(i_0)) *
        dim(ρ(i_1)) ≥ WS)) then
            transposeFVIMatchSmall()
        else
            transposeOrthogonalArbitrary()
    else
        transposeOrthogonalArbitrary()
```

called the *Orthogonal-Distinct* schema. This case represents a direct generalization of the 2D matrix transposition scheme, where one or more contiguous fastest-varying dimensions for the input (output) tensor are combined to create a longer virtual dimension. For this scheme to be applicable, the set of combined tensor indices from the input tensor and the output tensor cannot overlap (this is elaborated upon later in this section); hence the label for this scheme. The figure depicts the transposition of $[i_0, i_1, i_2] \Rightarrow [i_2, i_1, i_0]$. Each thread block transposes a 9×7×64 slice of the input tensor (we explain later how slice-size is chosen) in 4 phases. In each phase, each thread block processes 32×32 elements using a 32×33 2D shared-memory buffer (phases dealing with elements at a slice-boundary may process fewer elements). Each phase consists of two steps: i) copying elements from the input tensor to shared memory, and ii) moving elements from shared memory to the output tensor.

Next, let us consider the tensor transposition $[a, b, c, d] \Rightarrow [c, b, d, a]$, with the extent of dimensions a, b, c, d being $8, 2, 8, 8$, respectively. Possible ways of combining indices for the Orthogonal-Distinct schema are: $[\{a, b\}, c, d] \Rightarrow [\{c\}, b, d, a]$ (slice size: 16×8), $[\{a\}, b, c, d] \Rightarrow [\{c, b\}, d, a]$ (slice size: 8×16), $[\{a\}, b, c, d] \Rightarrow [\{c, b, d\}, a]$ (slice size: 8×128). With any of these choices, at least one of the input/output tensors will involve a combined fastest-varying index smaller than 32, resulting in inefficient data transfer to/from global-memory.

However, let us consider the combining of indices as: $[\{a, b, c\}, d] \Rightarrow [\{c, b, d\}, a]$. Now the combined index for the input tensor and the combined index for the output tensor both have a fused size of 128 each. But such a combining involves some common indices on both the input and output tensor. This means that the previous 2D orthogonal view for the Orthogonal-Distinct schema does not hold, i.e., the overlapped indices cannot be mapped both to the columns as well as the rows of a 2D shared-memory buffer. However, as explained later with algorithmic details, it is feasible to achieve coalesced global-memory accesses both for copy-in from the input-tensor as well as copy-out to the output-tensor, by using indirection arrays to address the data elements in the shared-memory buffer. Instead of the simple row-wise and column-wise access of data in the shared-memory buffer, more complex data access patterns are required, but can be achieved efficiently because of the flexibility in accessing data from shared-memory. This schema is labeled *Orthogonal-Arbitrary*.

In addition to the two cases discussed above, TTLG implements specialized data movement strategies for the case where the fastest varying index (FVI) is the same for both input and output tensors, e.g., $[a, b, c, d] \Rightarrow [a, d, c, b]$. For the FVI_Match case, if the FVI extent is greater than or equal to 32, a direct copying strategy is performed, without any need to buffer data in GPU shared-memory. If FVI extent is less than 32, a shared-memory buffering strategy is used, where a set of contiguous segments of size —FVI— are copied into a shared-memory buffer and then copied out to the output tensor.

A high-level decision flow-chart for choice of the different schema is shown in Fig. 3, and also shown as pseudocode

Algorithm 2: Orthogonal-Distinct

Input : $in\{i_0, i_1, \cdots, i_{d-1}\}$: d-dimensional input tensor
$dimsIn$: Selected set of dimensions for input slice [Alg. 3]
$dimsOut$: Selected set of dimensions for output slice[Alg. 3]
$blockA$: Selected blocking factor for the slowest varying dimension in the input slice [Alg. 3]
$blockB$: Selected blocking factor for the slowest varying dimension in the output slice [Alg. 3]
in_offset : input offsets [Alg. 4]
out_offset : output offsets [Alg. 4]
ρ: Permutation function, maps indices from input to output
Output: $out\{\rho(i_0), \rho(i_1), \cdots, \rho(i_{d-1})\}$: d-dimensional output tensor

```
1  kernel transposeOrthogonalDistinct()
2      vol_input_slice = volume(dimsIn, blockA)
3      vol_output_slice = volume(dimsOut, blockB)
4      coars_rows = ceil(vol_input_slice/WS)
5      coars_cols = ceil(vol_output_slice/WS)
6      [i_p, ..., o_q, ...] = decode(blockid)
7      [in_base, out_base] = compute_base(i_p, ..., o_q, ...)
8      out_row = threadid%WS
9      in_col = threadid%WS
10     for row_tile = 0 to coars_rows do
11         start_row = threadid/WS
12         start_col = threadid/WS
13         for col_tile = 0 to coars_cols do
14             for
               in_row = start_row to WS step threadblksize/WS
               do
15                 sm_buf[in_row][in_col] =
                   in_base[in_offset[in_row + row_tile * WS] +
                   in_col + col_tile * WS]
16             __syncthreads()
17             for out_col = start_col to WS step
               threadblksize/WS do
18                 out_base[out_offset[out_col + col_tile * WS] +
                   out_row + row_tile * WS] =
                   sm_buf[out_row][out_col]
19             __syncthreads()
```

in Alg. 1. In Figure 3, index fusion refers to fusing the indices that occur consecutively both in the input and in the output tensors. For example, if $[i_0, i_1, i_2, i_3] \Rightarrow [i_3, i_1, i_2, i_0]$ is the transposition, after fusing the consecutive indices i_1 and i_2, we consider the transposition as $[i'_0, i'_1, i'_2] \Rightarrow [i'_2, i'_1, i'_0]$, where size$(i'_1)$ = size(i_1) $\times$ size(i_2), size(i'_0) = size(i_0) and size(i'_2) = size(i_3). In the decision flowchart, first the FVI on input and output tensor are compared (shown using the permutation function in the figure). If they match, the extent of FVI is compared with 32. If larger, direct transfer without shared-memory is used (Alg. 7). If FVI extent is smaller than 32, but the product of extents of the two fastest varying indices exceeds 32, a specialized shared-buffering strategy is used (Alg. 6); if not, the Orthogonal-Arbitrary scheme (Alg. 4) or Alg. 6 may be used (based on performance prediction). On the left half of the flowchart, we have the case of non-matching FVI, where transposition is performed using either the Orthogonal-Distinct (Alg. 2) or the Orthogonal-Arbitrary (Alg. 4) strategy.

IV. Algorithmic Details for Tensor Transposition

As described in Section III, to perform a transpose operation, we combine multiple dimensions starting from the fastest varying dimension (in both input and output tensor) until we obtain a sufficient number of contiguous elements to achieve efficient coalesced access for data movement from/to global memory. Algorithm 2 presents details for the Orthogonal-Distinct case. For this algorithm to be applicable, there should be no common indices among the selected FVI indices (for combining) from the input and output tensors. For example, consider the transpose operation $[a, b, c, d] \Rightarrow [d, c, b, a]$. If we combine indices a, b, c on the input and combine indices d, c on the output, index c would be included on both the input and output tensors, making it inapplicable for the Orthogonal-Distinct case. We refer to the product of extents of selected indices as the volume of the input slice and output slice. We use Algorithm 3 to get the selected indices for the input and output slices (shown as $dimsIn$ and $dimsOut$ in the Input of Alg. 2). Algorithm 3 uses performance modeling to predict the execution times and chooses the indices for the slices (described in detail later). Once the slice sizes are chosen at Line 2 and at Line 3, each data slice is assigned to a thread block for transposition.

For each data transfer, the kernel needs to compute offsets for the global-memory as well as for the shared-memory, using indices of all the dimensions of the input tensor and the output tensor – requires expensive mod and div operations. The kernel calculates for each thread block, the base addresses corresponding to the outer indices of the tensors, and the offset addresses corresponding to the inner indices of the tensors. The `decode` function at Line 6 performs mod and div operations on blockid (refers to the built-in CUDA variable blockIdx.x) to get the outermost indices of the tensors and the `compute_base` function computes the base addresses (at Line 7) using these indices and the dimension sizes (not shown in the parameters). Offset addresses are precomputed in input and output *offset arrays* with the stride values corresponding to input and output slice dimensions (computed by Algorithm 4, see Input of Algorithm 2). We note that the offset arrays are common for all the slices. Since the offset indexed arrays are invariant across the slices and shared by all the thread blocks, there is high reuse. Hence these arrays are mapped to texture memory. In practice, cache hit rates for the offset arrays are generally greater than 99%.

For the Orthogonal-Distinct case, a fixed shared memory buffer of size of $\mathcal{WS} \times (\mathcal{WS} + pad)$, i.e., 32×33, is used (one extra column is used as padding to avoid bank conflicts). If the volume of the data slice handled by a thread block is higher than the shared memory capacity, thread coarsening is used. A sub-slice of size $\mathcal{WS} \times \mathcal{WS}$, (32×32) is copied in from the input tensor to shared memory at Line 15 and then copied out to the output tensor at Line 18. This is done in multiple iterations until the complete slice is processed. The warp size (32) is used for determining the shared memory size and the sub-slice size; the advantages associated with this are: (1) As the warp size is 32, if each warp brings in 32 elements or its multiple, we can expect to achieve 100% warp efficiency; (2) Since the transaction size is 128 bytes, all the 32 elements can be moved in a single transaction in case of `float` (two transactions in case of `double`), utilizing the complete transaction(s) in both the cases; (3) As the maximum shared memory limit per SM is 48KB, using it for storing 32*33 `float` or `double` values ensures good occupancy. All the read and write accesses are fully coalesced for this algorithm and there are no bank conflicts, enabling

Algorithm 3: Slice size choice for Orthogonal-Distinct

Input : I: Set of indices for the input tensor, $\{i_0, i_1, \ldots, i_{d-1}\}$
O: Set of indices for the ouput tensor, $\{o_0, o_1, \ldots, o_{d-1}\}$
$SharedMemLimitPerSM$: Shared memory size of a SM
$overbooking\_factor$: Multiplication factor for overbooking the threadblocks (empirical)
Output: $blockA$: Selected blocking factor for the slowest varying dimension in the input slice
$blockB$: Selected blocking factor for the slowest varying dimension in the output slice
$dimsIn$: Selected set of dimensions for the input slice
$dimsOut$: Selected set of dimensions for the output slice

1 **function** `slice_size_choice()`
2 $volume\_input = \Pi_{k=0}^{d-1} R_{i_k}$
3 min_num_blocks = num_SM $*$ SharedMemLimitPerSM / shared_memory_used_per_block // 32×33
4 $maxlimit$ = volume_input / (overbooking_factor $*$ min_num_blocks $*$ shared_memory_used_per_block)
5 $bestTime = \infty$
6 **for** $limitir = \mathcal{WS}$ **to** $maxlimit$ **step** $\mathcal{WS}$ **do**
7 **for** $limitor = \mathcal{WS}$ **to** $maxlimit/limitir$ **step** $\mathcal{WS}$ **do**
8 **for** $k = 1$ **to** $|I|$ **do**
9 **if** $\Pi_{l=0}^{k} R_{i_l} \geq limitir$ **then**
10 $blockA = ceil(limitir / \Pi_{l=0}^{k-1} R_{i_l})$
11 $dims\_input\_slice = I - \{i_{k+1}, \ldots, i_{d-1}\}$
12 *break*
13 **for** $k = 1$ **to** $|O|$ **do**
14 $dims\_output\_slice = O - \{o_{k+1}, \ldots, o_{d-1}\}$
15 **if** *(dims_input_slice* $\cap$ *dims_output_slice)* $\neq \emptyset$ **then**
16 *break*
17 **if** $\Pi_{l=0}^{k} R_{\rho^{-1}(o_l)} \geq limitor$ **then**
18 $blockB = ceil(limitor / \Pi_{l=0}^{k-1} R_{\rho^{-1}(o_l)})$
19 $time$ = get_PredictedTime($\Pi_{k=0}^{blockAdim-1} R_{i_k}$, $\Pi_{k=0}^{blockBdim-1} R_{\rho^{-1}(o_k)}$, blockA, blockB, blockAdim, blockBdim)
20 **if** $time \leq bestTime$ **then**
21 $bestTime = time$
22 $best\_dims\_in = dims\_input\_slice$
23 $best\_dims\_out = dims\_output\_slice$
24 $best\_blockA = blockA$
25 $best\_blockB = blockB$
26 *break*
27 $dimsIn = best\_dims\_in$; $dimsOut = best\_dims\_out$
28 $blockA = best\_blockA$; $blockB = best\_blockB$

Algorithm 4: Offset calculation for Orthogonal-Arbitrary

Input : $in\{i_0, \cdots, i_{d-1}\}$: d-dimensional input tensor
I : Set of indices for the input tensor, $\{i_0, i_1, \cdots, i_{d-1}\}$
O : Set of indices for the output tensor, $\{o_0, o_1, \cdots, o_{d-1}\}$
$dimsIn$: Selected set of dimensions for input slice [Alg. 3]
$dimsOut$: Selected set of dimensions for output slice[Alg. 3]
$perm$: Permutation of output indices in terms of input indices
Output: $input\_offset$: Slice offset for input
$output\_offset$: Slice offset for output
sm_out_offset : Slice offset in buffer (shared memory) corresponding to output

1 **function** `Offset_Calculation()`
2 $dimsOnlyOut = dimsOut - dimsIn$
3 $ilimit =$ `volume`$(dimsIn)$
4 $olimit =$ `volume`$(dimsOnlyOut)$
5 **for** *rowId = 0* **to** *olimit* **do**
6 $index = rowId$
7 $offseti = 0$
8 **for** *iter = 0* **to** `num_dims` *(dimsOnlyOut)* **do**
9 $dval = dimsOnlyOut[iter]$;
10 $offseti$ += $(index \% dval) *$ `get_stride`(I, $perm[dimsOnlyOut[iter]]$);
11 $index = index / dval$;
12 **input_offset**$[rowId] = offseti$;
13 **for** *colId = 0* **to** `num_dims` *(dimsIn)* **do**
14 $buffoff =$ `decode`$(rowId * ilimit + colId, dimsOut,$ `get_stride`$(dimsIn + dimsOnlyOut,$ $permute(dimsOut), perm))$;
15 **sm_out_offset**$[rowId][colId] = buffoff$;
16 $offseto =$ `decode`$(rowId * ilimit + colId,$ $dimsOut,$ `get_stride`(O, $permute(dimsOut), perm))$;
17 **output_offset**$[rowId][colId] = offseto$;

good performance of this kernel. Pre-computation of the offset arrays – by avoiding expensive mod and div operations in the inner-most loops – further contributes to high achieved performance.

Algorithm 3 shows how slice sizes are chosen for Algorithm 2. We use the performance model discussed in Section V to choose the slice sizes with the best predicted performance. Here, all slices of volume $A \times B$ are considered, where A and B are the combined dimension lengths along the input and output tensors from the fastest varying dimension, and both A and B are the minimum value above some multiple of $\mathcal{WS}$ (two outer loops in the algorithm move in steps of $\mathcal{WS}$). To get the minimum value above some multiple of $\mathcal{WS}$, we also consider possible blocking of a dimension (the slowest varying dimension in the group is blocked – Line 10 and Line 18). If each warp brings in warp size (or multiple) number of elements, we can achieve good warp efficiency. The maximum slice size is limited to ensure sufficiently high thread block count for good occupancy. If $slice\_vol = input\_slice\_size \times output\_slice\_size$, in Algorithm 2, the number of elements moved by each thread block is $slice\_vol$; higher the $slice\_vol$, lower the total number of thread blocks. It is important to have a sufficient number of thread blocks to occupy all the SMs (Streaming Multiprocessors) in a GPU to achieve good performance. We empirically calculated a value – $overbooking\_factor$ at Line 4 – that when multiplied with min_num_blocks, gives a sufficient number of thread blocks, based on which the upper bound ($maxlimit$) for a slice size is calculated. For each admissible slice size, its performance is estimated using the performance evaluation model at Line 19 (explained in Section V).

Algorithm 2 requires no common indices between the selected indices in the input and the output. Algorithm 5 is used when there is at least one common index among the combined FVI indices of the input and output tensors. Similar to the algorithm for the Orthogonal-Distinct case (Algorithm 2), Algorithm 5 also uses offset arrays to store the strides along the dimensions mapped to rows and columns of the shared memory. Algorithm 4 shows the computation of these offset arrays. If $IS = \{i_1, \ldots i_x\}$ is a set of input slice indices and $OS = \{o_1, \ldots o_y\}$ is a set of output slice indices, $OOS = OS - IS$ is a set of indices that are present in the output slice but not in the input slice (in the algorithm, we refer to IS as $dimsIn$, OS as $dimsOut$ and OOS as $dimsOnlyOut$). The shared memory buffer can be viewed as 2D, with indices from the set IS on columns and with indices from the set OOS on rows. In Algorithm 4, in each iteration, at Line 9, $dval$ takes the size of a index i such that $i \in OOS$.

Algorithm 5: Orthogonal-Arbitrary

Input : $in\{i_0, i_1, \cdots, i_{d-1}\}$: d-dimensional input tensor
$dimsIn$: Selected set of dimensions for input slice [Alg. 3]
$dimsOut$: Selected set of dimensions for output slice[Alg. 3]
$blockA$: Selected blocking factor for the slowest varying dimension in the input slice [Alg. 3]
$blockB$: Selected blocking factor for the slowest varying dimension in the output slice [Alg. 3]
in_offset : input offsets [Alg. 4]
out_offset : output offsets [Alg. 4]
sm_out_offset : shared memory output offsets [Alg. 4]
ρ: Permutation function, maps indices from input to output
Output: $out\{\rho(i_0), \rho(i_1), \cdots, \rho(i_{d-1})\}$: d-dimensional output tensor

1 **kernel** `transposeOrthogonalArbitrary()`
2 $inp\_vol =$ `volume` (*dimsIn, blockA*)
3 $out\_vol =$ `volume` (*dimsOut - dimsIn, blockB*)
4 $[i_p, \cdots, o_q, \cdots] =$ `decode` (*blockid*)
5 $[in\_base, out\_base] =$ `compute_base` $(i_p, \cdots, o_q, \cdots)$
6 $row\_incr =$ `ceil` (*inp_vol*) $/threadblocksize$
7 $start\_col = threadid \% inp\_vol$
8 $start\_row = threadid / inp\_vol$
9 **for** $row = start\_row$ **to** out_vol **step** row_incr **do**
10 **for** $col = start\_col$ **to** in_vol **step** $threadblocksize$ **do**
11 $sm\_buff[row * inp\_vol + col] = in[in\_offset[row] + col]$
12 `__syncthreads()`
13 **for** $row = start\_row$ **to** out_vol **step** row_incr **do**
14 **for** $col = start\_col$ **to** in_vol **step** $threadblocksize$ **do**
15 $out[out\_offset[row * inp\_vol + col]] = sm\_buff[sm\_out\_offset[row * inp\_vol + col]]$

The algorithm computes the starting address of each row using every $dval$ and stores it in the $input\_offset$ array at Line 12. Similarly, the algorithm computes the offsets of every element in the IS by OOS array corresponding to shared memory and output tensor (global memory) into sm_out_offset array at Line 15 and $output\_offset$ array at Line 17, respectively.

Algorithm 5 presents pseudocode for the Orthogonal-Arbitrary case. Here, each thread initially adds the base address of the slice to the corresponding input offset to find the first element to be copied. Thereafter, each thread adds the row length to find the next element to be copied, thus avoiding the expensive mod and div operations to compute the address of the element (at Line 11). Note that this optimization is enabled by the property that all the dimensions being mapped to the slice are in the input order (contiguous). However, this optimization cannot be applied to the output because the dimensions being mapped to the slice might not be in same order as in the output tensor. Hence, an output offset array is used to point to each element of the slice (at Line 15). We also need an offset array for the shared memory read access (i.e. while writing to the output tensor) as there is no regular pattern in the way elements get mapped to the shared memory – the pattern depends on the given permutation and sizes of the dimensions. On the input side we are guaranteed full coalescing, but on the output side the accesses may have breaks in between, if the dimensions being mapped to the slice are not consecutive in the output tensor.

Even though Algorithm 5 can handle all the tensor transposition cases, it has the additional overhead of offset arrays. Also, it could suffer from some shared memory bank conflict. However, these issues can be solved by specialization in many cases.

Algorithm 6: FVI-Match-Small

Input : $in\{i_0, i_1, \cdots, i_{d-1}\}$: d-dimensional input tensor
ρ: Permutation function, maps indices from input to output
Output: $out\{\rho(i_0), \rho(i_1), \cdots, \rho(i_{d-1})\}$: d-dimensional output tensor

1 **kernel** `transposeFVIMatchSmall()`
2 $N_0 = dim(i_0)$
3 $[i_k\_lb, i_1\_lb] =$ `decode` (*blockid*)
4 $warp_{id} = threadid / \mathcal{WS}$
// Copy in a chunk of $in[i_0, i_1 : b, \cdots, i_k : b, \cdots]$ to shared memory buffer in coalesced manner
5 **for** $iter = i_1\_lb + threadid$ **to** $(i_1\_lb + N_0 * b)$ **step** $\mathcal{WS}$ **do**
6 $sm\_buf[warp_{id}][iter] = in[threadid \% \mathcal{WS}][iter]$
7 `__syncthreads()`
// Copy from shared memory to $out[i_0, i_k : b, \cdots, i\_1 : b, \cdots]$ in coalesced manner
8 **for** $iter_1 = i_k\_lb + threadid$ **to** $(i_k\_lb + N_0 * b)$ **step** $\mathcal{WS}$ **do**
9 $out[threadid \% \mathcal{WS}][iter_1] = sm\_buf[threadid / N_0][threadid \% N_0 + warp_{id} * N_0]$

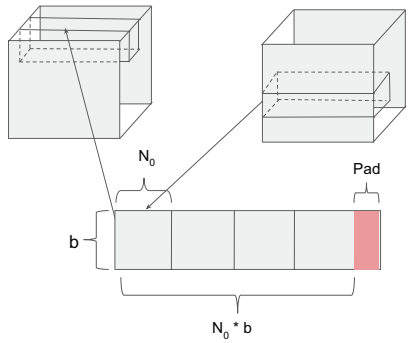


Fig. 4: Illustration of approach for FVI-Match-Small case.

Algorithm 6 presents pseudocode for the approach to efficient transposition of tensors with small matching fastest varying index (FVI). The scheme is illustrated in Fig. 4. In order to achieve high warp efficiency as well as coalesced access on both reading from and writing to global memory, data is moved in slices that can be logically mapped to a 3D block of size $b \times b \times N_0$ (b is the blocking factor on indices next-to fastest varying index on input and output, whose size is chosen by performing modeling). The threads in a warp collectively move a bundle of b consecutive rows along i_1, while other warps in the thread block concurrently move other contiguous chunks of size $b \times N_0$ for adjacent values of i_k, where i_k is the index of the second fastest varying index in the output tensor. After the $b \times b \times N_0$ elements are copied into a shared memory buffer at Line 6, they are then collectively written from the buffer to the output tensor in transposed form at Line 9. This is done by having each warp now collect a bundle of b 'pencils' along the orthogonal dimension and writing a contiguous chunk of $b \times N_0$ words in the output tensor.

Viewing the $b \times b \times N_0$ 3D shared-memory buffer as an equivalent 2D array with b rows and bN_0 columns, as shown in Fig. 4, suitable padding is introduced to ensure that threads in a warp can access a collection of vertically stacked 'pencils' without any bank conflicts. Let us assume that the first N_0 elements in the first row of the buffer map to banks

Algorithm 7: FVI-Match-Large

Input : $in\{i_0, i_1, \cdots, i_{d-1}\}$: d-dimensional input tensor
ρ: Permutation function, maps indices from input to output
Output: $out\{\rho(i_0), \rho(i_1), \cdots, \rho(i_{d-1})\}$: d-dimensional output tensor

```
1 kernel transposeFVIMatchLarge()
2     [i_p, i_1] = decode (blockid)
3     [in_base, out_base] = compute_base (i_p, i_1) ;      // ρ(i_p)
      and ρ(i_1) are used to compute out_base.
4     for iter = threadid to N_0 step threadblocksize do
5         out_base[iter] = in_base[iter]
```

$0, 1, \cdots N_0 - 1$, the value of pad is chosen so that element 0 in row 1 of the 2D view of the shared-memory buffer maps to memory bank N_0. This ensures completely conflict-free access for the write-out phase.

Finally, Algorithm 7 shows pseudocode for the case of matching FVI but range greater than $\mathcal{WS}$. For this case, no shared memory is used, but contiguous elements are just read in to registers and written out in a coalesced manner by threads of a warp.

A. Coarsening

For all the algorithms except the Orthogonal-Distinct case, the slice size is chosen based on the shared memory size used by a thread block. As discussed in Algorithm 2 and Algorithm 5, for each slice, a base address calculation with costly mod and div operations is required. This cost can be alleviated by using thread coarsening. We apply a heuristic to possibly consider one dimension for coarsening. The slice size gets multiplied by the size of the coarsening dimension (if any). The expensive base address computation using mods and divs is only performed for the first sub slice. For subsequent sub slices, the base addresses (input and output) are obtained by adding the strides corresponding to the coarsened dimension to the base addresses of the previous sub slice. The heuristic we use to select the coarsening dimension is to consider the first dimension in the input order from the fastest varying one with extent between 4 and 32. A very high coarsening factor may reduce the total number of thread blocks and may affect occupancy and cause tail effects. In order to avoid this problem, we only consider coarsening for tensors of sizes greater that 2MB.

B. Register Usage

Modern GPUs map small arrays to registers when the index can be statically determined to be constant. To make use of this we ensure constant indexing to the dimension and the stride arrays in the code. We do this by writing macro defined code corresponding to the tensor rank (until rank 15). Since dimension-blocking is possible on the input and output tensor, we need to index into the arrays corresponding to their index locations to identify the full and partial blocks. If this checking is done via variables (as blocking dimension index varies with respect to input), it could inhibit the register usage of the arrays. In order to handle this, we always move the two blocking dimensions to the beginning of the arrays and index them by 0 and 1, and consider the rest of the dimensions from index 2 onwards. This way, all the array accesses required for slice base calculation and for checking the full and partial blocks are fully compile time determinable constants.

Algorithm	Input			Output		
	DRAM	SM	TM	DRAM	SM	TM
FVI-Match-Small	C_1	C_1	0	C_1	C_1	0
FVI-Match-Large	C_2	0	0	C_2	0	0
Orthogonal-Distinct	C_3	C_3	C_3	C_3'	C_3'	C_3'
Orthogonal-Arbitrary	C_3	C_3	C_3	C_3'	C_3'	$2 \times C_3'$

TABLE I: Analysis of all the four proposed algorithms.

C. Algorithm Analysis

In this sub-section, we quantify the data movement required for the various cases. For transposition $[i_0, i_1, \ldots, i_{n-1}] \rightarrow [o_0, o_1, \ldots, o_{n-1}]$, we quantify the number of load transactions and store transactions for each type of memory: DRAM, shared memory (SM) and texture memory (TM, used for offset arrays). We consider the 128 byte GPU transactions that can move 32 float elements at a time. For the FVI-Match-Small (Alg. 6) case, if b is the blocking factor on i_1 and o_1 (next indices to FVI), the number of DRAM load transactions is $C_1 = \lceil \frac{size(i_0) \times b}{32} \rceil \times \frac{\prod_{k=1}^{n-1} size(i_k)}{b}$, because $b \times size(i_0)$ *contiguous* elements are moved by a single warp and $\frac{\prod_{k=1}^{n-1} size(i_k)}{b}$ is the total number of warps. We can see from Alg. 6 that the #DRAM store transactions, #SM store transactions and #SM load transactions are the same as C_1. Table I shows the analysis for each of the proposed algorithms. For the FVI-Match-Large (Alg. 7) case, the number of DRAM load transactions is $C_2 = \lceil \frac{size(i_0)}{32} \rceil \times \prod_{k=1}^{n-1} size(i_k)$. Here, the left operand of the product ($\times$) shows the number of contiguous elements moved by a thread block and the right operand shows the number of thread blocks, and there is no use of shared memory. Similarly, for the Orthogonal-Distinct (Alg. 2) and Orthogonal-Arbitrary (Alg. 5) cases, if $IS = \{i_0, \ldots i_x\}$ is a set of input slice indices and $OS = \{o_0, \ldots o_y\}$ is a set of output slice indices, the number of DRAM load and store transactions is $C_3 = \left\lceil \frac{(\prod_{p=0}^{x-1} size(i_p)) \times b_x}{32} \right\rceil \times \frac{\prod_{k=x}^{n-1} size(i_k)}{b_x}$, b_x is the blocking factor on i_x and $C_3' = \left\lceil \frac{(\prod_{q=0}^{y-1} size(o_q)) \times b_y}{32} \right\rceil \times \frac{\prod_{k=y}^{n-1} size(o_k)}{b_y}$, b_y is the blocking factor on o_y, respectively. Here, along with DRAM and SM transactions, we have TM transactions used for the offset arrays. The last row of Table I shows the TM transactions corresponding to input_offset, output_offset and sm_out_offset arrays of Alg 4. We note that for ease of presentation we have assumed that a blocking dimension size is a perfect multiple of its blocking factor (computation for the remainder blocks is ignored).

V. Performance Modeling of Transpose

As discussed in the preceding sections, the transpose operation for some permutations can be performed by more than one algorithm, and multiple choices may exist for parameters like slice sizes and block sizes for each algorithm. For example, both Algorithm 2 and Algorithm 5 can be used for a transpose operation in which the fastest-varying-index of the input tensor does not match that of the output tensor; and for both these algorithms, slice and block sizes must be chosen. To address this, we use a performance model based on linear regression to predict the execution times for each of the proposed algorithms for different parameters. The comparative analysis in Section IV-C is not sufficiently detailed for accurate performance modeling. Instead we use additional parameters

to construct a linear regression model for each kernel. Due to space constraints, we only present modeling details for the more complex kernels, and omit modeling details for the FVI-Match-Small and FVI-Match-Large cases. We first describe the dataset used for constructing the models.

DataSet. To create a diverse dataset, we consider several transpose test cases that cover different ranks, volumes, extents (sizes of dimensions) and orderings among the extents. Tensor ranks range from 3 to 6 and include all possible permutations. We note that ranks 1 and 2 are also covered by this range when extents get combined with index fusion. Volumes of the tensors range from 16MB to 2GB. Different orderings among extents include (example shown in brackets for a 3D tensor with indices i_0, i_2, i_3): (1) all same ($i_0 = i_1 = i_2$) , (2) monotonically increasing ($i_0 < i_1 < i_2$), (3) monotonically decreasing ($i_0 > i_1 > i_2$), (4) increasing till the center dimension and then decreasing ($i_0 < i_1 > i_2$), (5) decreasing till the center dimension and then increasing ($i_0 > i_1 < i_2$). We randomly select four-fifths of all the configurations to form training data and the remaining configurations as test data.

Features. A key step in the modeling is the identification of appropriate parameters that capture the performance variation of the kernel. We now discuss the features used for modeling the Orthogonal-Distinct and Orthogonal-Arbitrary kernels (see Table II). The total execution time of any transpose is directly related to the amount of data moved. Thus, data volume is selected as a feature for both kernels. For these models, we also use input slice volume, output slice volume, and total number of threads (#ThreadBlocks*ThreadBlockSize) as features in the model.

As seen for the Orthogonal-Distinct kernel, the data slice processed by a warp, {input_slice X output_slice}, is viewed as a 2D space, and the transpose operation in the inner loop happens on each 32×32 tile of the 2D space. Since the input_slice an doutput_slice may not be perfect multiples of the warpsize, warp-level inefficiency due to idle threads will occur with "boundary" tiles. An abstract measure of number of "cycles" is computed to capture the inefficiency due to idle threads in warps.

We calculate the number of cycles spent on input and output for each full tile and the partial tiles. Let the number of full tiles and the count of the three possible kinds of partial tiles (remainder on input slice only, remainder on output slice only, and remainder on both input and output slices) be n_1, n_2, n_3, n_4, respectively, and the remainder tiles on input slice and on output slice be rem_1 and rem_2 respectively. Number of cycles spent is calculated as $f_1 = n_1 \times (32 + 32) + n_2 \times (32 + rem_2) + n_3 \times (rem_1 + 32) + n_4 \times (rem_1 + rem_2)$. With respect to the blocking performed on one of the indices on the input slice and one of the indices on the output slice, when the extent of the tensor is not a perfect multiple of the block-size for a partially traversed index in a slice, there will be partial slices similar to the partial tiles discussed above.. If the number of full and partial slices are N_1, N_2, N_3 and N_4; and f_1, f_2, f_3 and f_4 are the cycles calculated for full and partial slices, then the total number of cycles is computed as $N_1 \times f_1 + N_2 \times f_2 + N_3 \times f_3 + N_4 \times f_4$.

For the Orthogonal-Arbitrary kernel, we compute the cycles in terms of the number of transactions on the input and on the output side. Similar to the Orthogonal-Distinct case, we can have partial input and partial output slices based on the chosen blocking factors. If f_1 (=$C_3 + C'_3$, from Section IV-C) is the number of transactions on the full slices, and f_2, f_3 and f_4 are the number of transactions on the three types of partial slices, the number of cycles is calculated as $f_1 + f_2 + f_3 + f_4$. Here, as the threads move the elements of a slice in a round-robin fashion (unlike row/columns-wise in Orthogonal-Distinct), we consider the size of the contiguous memory chunks on the input and on output side in a slice as features, referred to as input stride and output stride, respectively. Special instructions here represent the number of integer mod and div operations, used for boundary checking in the remainder code (not shown in Algorithm 5). These instructions are automatically converted to floating point (MUFU) instructions by the compiler.

Algorithm	Feature	Estimate	Std. Error	t value	Pr($>\|t\|$)
Orthogonal-Distinct	Volume	1.278e-11	5.097e-15	2508.13	<2e-16
	NumBlocks	5.001e-08	9.926e-11	503.88	<2e-16
	Input slice	7.835e-07	9.369e-09	83.62	<2e-16
	Output slice	1.252e-06	9.355e-09	133.81	<2e-16
	Cycles	-2.692e-11	1.222e-12	-22.03	<2e-16
Orthogonal-Arbitrary	Volume	-3.018e-11	3.189e-12	-9.465	<2e-16
	NumThreads	2.730e-10	6.492e-12	42.058	<2e-16
	Total Slice	2.126e-07	2.086e-08	10.196	<2e-16
	Input Stride	-8.880e-12	7.295e-13	-12.173	<2e-16
	Output Stride	-1.091e-11	7.344e-13	-14.851	<2e-16
	Special Instr	3.047e-11	2.138e-12	14.246	<2e-16
	Cycles	5.112e-10	2.439e-11	20.962	<2e-16

TABLE II: Parameters and coefficients of linear regression fits for the Orthogonal-Distinct and Orthogonal-Arbitrary cases.

Model. We use linear regression to develop models for predicting the execution time of a given tensor transposition. The model for the Orthogonal-Distinct kernel was trained on 77,502 data points with different slice size configurations, and the Orthogonal-Arbitrary kernel was trained on 8042 data points (fewer configurations because in this case, the shared memory size is proportional to the slice volume, and the slice size configurations which crossed the shared memory limit are infeasible). Table II shows a summary of the two models. The last column in the table shows a low p-value (close to 0 and much less than the standard cut-off of 0.05) for all selected features, implying that all these features are significant in the two models. We calculate the precision of a model as mean((abs($actualTime - predictedTime$))/$actualTime$) $\times$ 100 and the error percentages on train and test data are: 1) Orthogonal-Distinct case: 4.161% and 4.159% respectively, and 2) Orthogonal-Arbitrary case: 11.084% and 10.75% respectively. We use these models to choose the right slice sizes among the possible ones which are expected to give the best performance. For the Orthogonal-Distinct case, Figure 5 shows the actual and predicted times for a transpose example with dimensions {27 27 27 27 27} and permutation '4 1 2 0 3' over 31 slice variants. The volume ($size\ of\ input\ slice \times size\ of\ output\ slice$) of different slice variants are shown on the X-axis and time is shown on Y-axis of the plot. The figure also highlights the best choice–the slice variant which

gives the best performance. We observe the predictions follow the trend of actual execution times. Using this model, we can choose the potential best slice variant (input slice size = 189 and output slice size = 27) for the kernel.

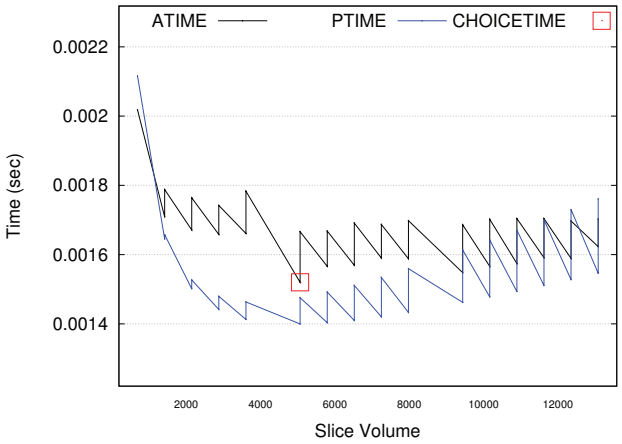


Fig. 5: Predictions of execution times for a transpose with dims: 27 27 27 27 27 and perm: 4 1 2 0 3 for Orthogonal-Distinct case.

VI. Experimental Evaluation

Resource	Details
CPU	Intel core i7-2600 (4 cores, 3.40 GHz, 8 MB L3cache, 16 GB DDR3-1333)
GPU	Tesla K40c (15 Kepler SMs, 192 cores/MP, 12 GB Global Memory, 745 MHz, 15 MB L2 cache, ECC off)
Software	Red Hat Enterprise Linux Server release 6.7x86 64, CUDA 7.5, GCC 4.8.1, Nvidia driver 352.79

TABLE III: Machine configuration

Table III shows the configuration of the machine used for evaluating TTLG. The following test cases were used:

- Transpositions involving all the $6! = 720$ possible permutations of a 6D tensor, with dimension sizes: (a) all of size 15, (b) all of size 16, and (c) all of size 17. Although the actual rank of the tensors is 6, due to index fusion, the effective tensor rank (labeled scaled rank in the performance charts) varies from 1 to 6.
- Transpositions for a fixed permutation but with varying dimension sizes/tensor volumes ranging over KBs, MBs and GBs
- The benchmark set [12] used by TTC [3] and cuTT [2], consisting of 57 input tensors of volume around 200 Mbytes and tensor ranks ranging from 2 to 6. The permutations chosen in the benchmarks are such that no index fusion is possible.

For each transposition, we report achieved $bandwidth$ (in GB/sec): given the $volume$ (product of dimension sizes) of the tensor and the $time$ taken for its transposition, $bandwidth = \frac{(2 \times volume \times 8)}{(time \times 10^9)}$. Each experiment was repeated five times and the average is reported. The variance across runs was extremely low for all experiments, with rarely a variation of more than 1% across the runs.

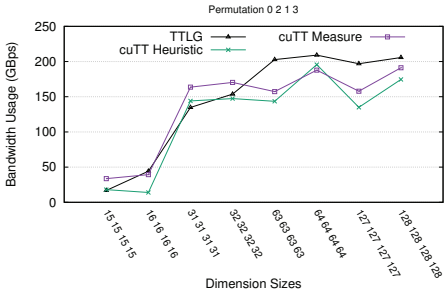


Fig. 13: Transpose Performance: Varying Dimension Sizes

We consider two use cases of a transpose library function: (1) a transpose operation is performed once or a small number of times on a tensor of specific shape/size, (2) a transpose operation is performed repeatedly a large number of times in an iterative application. TTC is a code generator and is targeted at the repeated use scenario. cuTT has a plan mode where an execution plan is created by first performing a number of measurements and saved for repeated use. cuTT also has a heuristic mode where an execution plan is chosen by a fast heuristic, with much lower overhead than the expensive measure operations for the plan mode. We compare performance with cuTT for both the single use case and the repeated use case.

Fig. 6, Fig. 8 and Fig. 10 present results for the repeated-use case with TTC, cuTT and TTLG, for all permutations of a 6D tensor, for sizes all 16, all 15, and all 17 respectively. The charts show achieved bandwidth for all 720 cases, grouped according to scaled rank of the tensor (the red "staircase" lines in the charts) after index fusion is performed. For example, for the permutation (0 2 1 3 4 6 5), the scaled rank of the tensor is 5 since 3,4 occur contiguously in the same order in the input and output tensor and can be fused. These results exclude any plan overhead (which includes memory allocation times for any buffer) and just corresponds to the kernel execution time. For most cases, TTLG outperforms cuTT-measure, which is always better than cuTT-heuristic. If any kernel is selected by cuTT heuristic, cuTT measure would also consider it, as the measure mode is elaborate, where all candidate kernels are executed and the best one is chosen. This way, due to possible cache effect in our experiments, cuTT measure timings had a very slight advantage as compared to cuTT heuristic even if the same kernel is chosen by both (visible at many points in the graph). TTC was found to be slower than the library based approaches.

Fig. 7, Fig. 9 and Fig. 11 present performance for the single-use scenario, thus including any plan creation time. TTLG's peak bandwidth has dropped from about 200 GBps to around 130 GBps and similarly for cuTT-heuristic. For cuTT-measure, the performance drop is much higher since its plan time includes multiple actual executions of the kernels. TTC is not shown in this graph, as it is a code generator and does not have an online plan time. Its offline code generation time took around 8 seconds for each input.

Fig. 12 presents performance for the repeated-use scenario

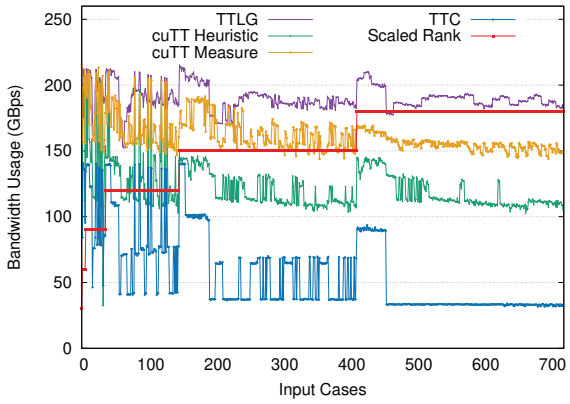


Fig. 6: Transpose of 6D Tensor (all 16) for Repeated Use Case

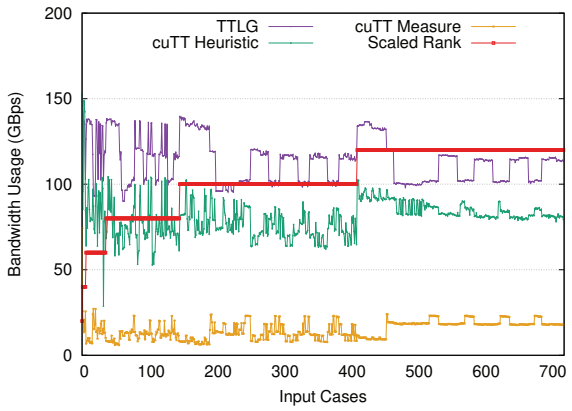


Fig. 7: Transpose of 6D Tensor (all 16) for Single Use Case

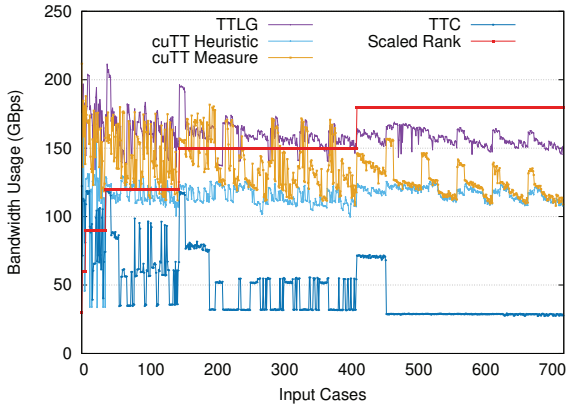


Fig. 8: Transpose of 6D Tensor (all 15) for Repeated Use Case

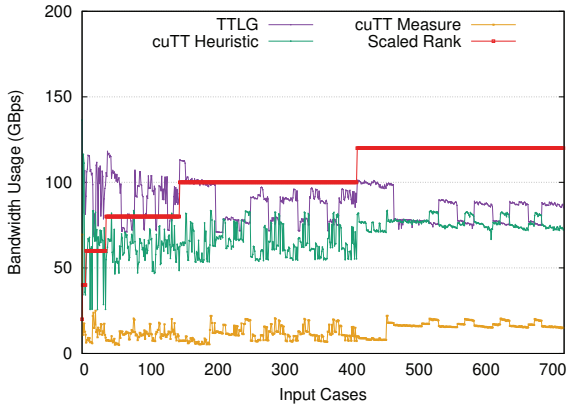


Fig. 9: Transpose of 6D Tensor (all 15) for Single Use Case

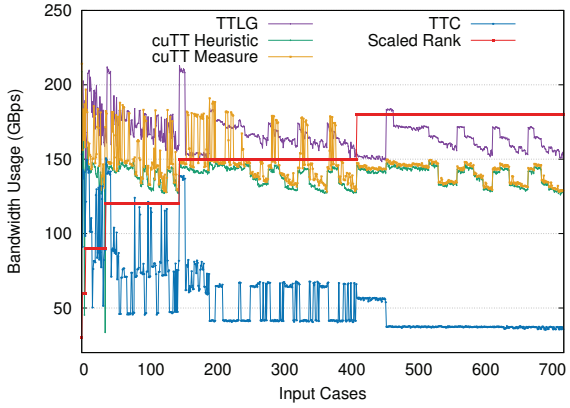


Fig. 10: Transpose of 6D Tensor (all 17) for Repeated Use Case

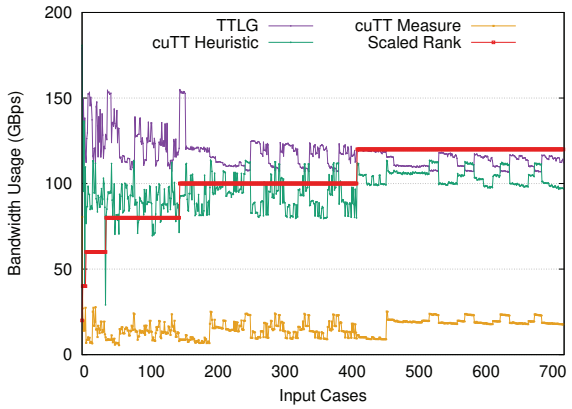
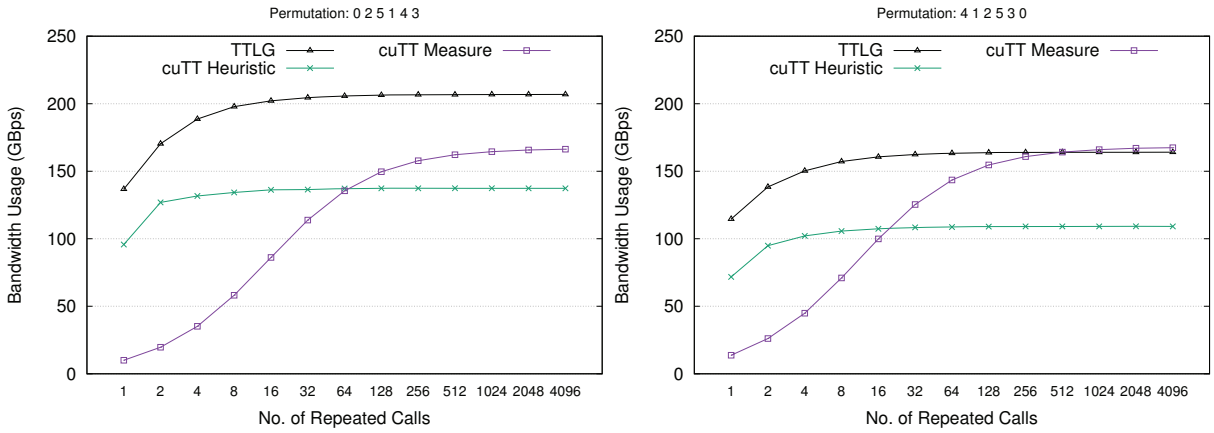


Fig. 11: Transpose of 6D Tensor (all 17) for Single Use Case

as a function of the number of repeated calls. We used a 6D tensor with all dimension sizes 16 and considered permutations P: '0 2 5 1 4 3' and '4 1 2 5 3 0', with fastest varying dimension at the left end. $P[i] = j$ means that the i^{th} dimension in the output corresponds to the j^{th} dimension in the input. For the '0 2 5 1 4 3' permutation (Fig. 12a), TTLG always performs better than cuTT-measure. For the '4 1 2 5 3 0' permutation (Fig. 12b), cuTT-measure eventually slightly outperforms TTLG after about 500 repeated calls. Fig. 13 shows the performance comparison for varying dimension sizes of the tensor. For smaller volumes, the achieved bandwidth is low for all the three library implementations. Once the volume is reasonably large, TTLG outperforms cuTT.

Fig. 14 presents performance comparison for the TTC



(a) Matching FVI

(b) Non-matching FVI

Fig. 12: Transpose Performance: Repeated Use Case

benchmarks. For most cases, TTLG outperforms cuTT-measure and cuTT-heuristic. Performance of TTC is much better for these inputs, but is still below both TTLG and cuTT.

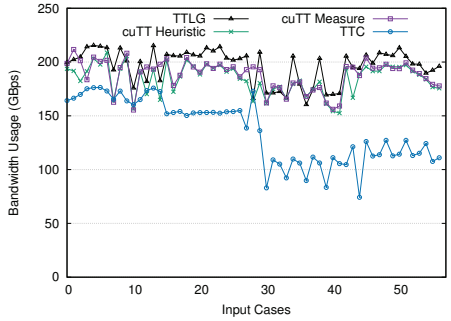


Fig. 14: Transpose Performance: TTC benchmarks

VII. Conclusion

In this paper, we have presented TTLG – an efficient library to perform tensor index-permutations (transposition) on GPUs. It is based on a systematic analysis of different permutation scenarios and development of four kernels to effectively handle the multitude of possible cases to enable efficient coalesced access of data to/from global memory. Further, the library incorporates a queryable performance prediction model that can be used by higher level libraries for devising optimized kernels that use tensor transposition as a building block. TTLG performs comparably or better than currently available alternatives for the repeated-use case and is considerably better for the single-use case.

Acknowledgments

We thank the anonymous reviewers for their feedback and suggestions that helped improve the paper. This work was supported in part by the U.S. National Science Foundation (NSF) through awards 1440749 and 1513120, by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under award number 71648, and 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. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830. Jyothi Vedurada is supported by the TCS Research Scholarship Program for her doctoral studies.

References

[1] D. I. Lyakh, "An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU," *CPC*, 2015.

[2] A. Hynninen and D. I. Lyakh, "cuTT: A High-Performance Tensor Transpose Library for CUDA Compatible GPUs," *CoRR*, 2017.

[3] P. Springer, A. Sankaran, and P. Bientinesi, "TTC: A Tensor Transposition Compiler for Multiple Architectures," in *ARRAY*, 2016.

[4] P. Springer, T. Su, and P. Bientinesi, "HPTT: A High-performance Tensor Transposition C++ Library," in *ARRAY*, 2017.

[5] Q. Lu, S. Krishnamoorthy, and P. Sadayappan, "Combining Analytical and Empirical Approaches in Tuning Matrix Transposition," in *PACT*, 2006.

[6] L. Wei and J. Mellor-Crummey, "Autotuning tensor transposition," in *IPDPS*, 2014.

[7] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *ISCA*, 2009.

[8] G. Mateescu, G. H. Bauer, and R. A. Fiedler, "Optimizing Matrix Transposes Using a POWER7 Cache Model and Explicit Prefetching," *ACM SIGMETRICS*, 2012.

[9] A. Derler, R. Zayer, H.-P. Seidel, and M. Steinberger, "Dynamic Scheduling for Efficient Hierarchical Sparse Matrix Operations on the GPU," in *ICS*, 2017.

[10] J. L. Jodra, I. Gurrutxaga, and J. Muguerza, "Efficient 3D Transpositions in Graphics Processing Units," *IJPP*, 2015.

[11] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988. [Online]. Available: http://doi.acm.org/10.1145/48529.48535

[12] P. Springer, "TTC benchmark," 2016, https://github.com/HPAC/TTC/blob/master/benchmark/benchmark.py.