Efficient Primitives for Standard Tensor Linear Algebra

David M. Rogers
University of South Florida
4202 E Fowler Avenue
Tampa, Florida 33620
davidrogers@usf.edu

ABSTRACT

This paper presents the design and implementation of low-level library to compute general sums and products over multi-dimensional arrays (tensors). Using only 3 low-level functions, the API at once generalizes core BLAS1-3 as well as eliminates the need for most tensor transpositions. Despite their relatively low operation count, we show that these transposition steps can become performance limiting in typical use cases for BLAS on tensors. The execution of the present API achieves peak performance on the same order of magnitude as for vendor-optimized GEMM by utilizing a code generator to output CUDA source code for all computational kernels. The outline for these kernels is a multi-dimensional generalization of the MAGMA BLAS matrix multiplication on GPUs. Separate transpositions steps can be skipped because every kernel allows arbitrary multi-dimensional transpositions of the arguments. The library, including its methodology and programming techniques, are made available in SLACK. Future improvements to the library include a high-level interface to translate directly from a $\text{\LaTeX}$ -like equation syntax to a data-parallel computation.

CCS Concepts

•Mathematics of computing → Computations on matrices; •Computing methodologies → Vector / streaming algorithms; •Software and its engineering → Source code generation;

Keywords

tensor; blas; transposition; gpu

1. INTRODUCTION AND BACKGROUND

Linear algebra on large arrays is central to scientific computing and all applied mathematical modeling. An increasingly large number of problem structures, however, do not fit well to the simple vector-or-matrix model presented by

basic linear algebra subprograms (BLAS)[7] – which form the currently accepted foundation for all dense matrix operations. Instead, most dense numerical algorithms fit much more naturally to arrays – a logical generalization including vectors (dimension 1) and matrices (dimension 2) as special cases. Their popularity can be easily seen by the thousands of packages based on the general array implementation in python’s *numpy* package.[20]

An n -dimensional array is defined as a block of data with n indices, $i_0, \dots, i_{n-1}$. Each index has a contiguous range, $i_k = 0, \dots, s_k - 1$, where $s = \{s_0, \dots, s_{n-1}\}$ is the shape of the array. The set of all arrays of shape s is denoted $\mathcal{T}_s$ (the terms tensor and array are used interchangeably). Here, we assume dense, non-symmetric arrays, which require memory space for $\prod_{k=0}^{n-1} s_k$ unique array elements.

Despite their popularity, there are no general alternatives to BLAS operations for multidimensional arrays. At a minimum, these operations should allow addition, multiplication, contraction, and transposition of arrays. The *numpy* implementation also provides several (map-like) per-element functions such as sine, cosine, and absolute values, as well as (reduce-like) single-index operations such as min/max/sum. Currently, *numpy*’s low-level implementation of these operations relies on a general tensor transposition implemented on the CPU combined with a translation to BLAS. It does not permit high-level abstraction or heterogeneous execution.

Standard linear array computational kernels (SLACK) is intended to be a functional replacement for BLAS that works efficiently with arrays. Source code created in the course of the present work has been made available in an open-source library (USF-SLACK) that implements this interface. One major result of this work has been to demonstrate that these kernels can be just as efficient as current high-performance BLAS implementations.

Several works have addressed the issue of generating code to efficiently compute tensor operations in parallel, but fall short of a generally useful library. The tensor contraction engine (TCE) translates a domain-specific language for computational quantum chemistry into a series of low-level FORTRAN loops for tensor summations and transpositions.[2] Much of the development effort for that code has gone into identifying high-level reorganizations of the modeling equations, and work is still underway to improve parallel execution methods.[1] The cyclops tensor framework (CTF)[19] is a similar project aimed at implementing efficient tensor contractions for large tensors stored in distributed global memory. Its interface uses essentially equivalent `contract` and `sum` functions, but specifies permutations by labeling

Permission to make digital or hard copies of all or part 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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

XSEDE16, July 17–21, 2016, Miami, FL, USA

© 2016 ACM. ISBN 978-1-4503-4755-6/16/07...\$15.00

DOI: <http://dx.doi.org/10.1145/2949550.2949580>

all indices on each input tensor using character strings and has non-standard convention for choosing where results are output.

One of the motivations for the present work is the observation stemming from these projects that tensor transposition accounts for nearly half the execution time of typical quantum chemistry problems.[4] The other is that a fast, standard library for contractions would greatly speed development time for implementing many related methods. As a case-in-point, there is already an N^4 scaling density-fitting method for carrying out CCSD calculations which otherwise scale as N^6 . [5] Unfortunately, that method has not yet been implemented because of the hundreds of contraction steps that require intense effort to convert down to simple, space-efficient BLAS operations.[3]

Similar observations have motivated development on the tensor contraction generator (TCG), [16, 10, 9] and other low-level SIMD coprocessor generators for tensor contractions.[17, 14] Unfortunately, many of these codes are not generally available. Performance results for TCG were only reported for double precision (shown to achieve a peak of around 40 Gflops), but not compared to vendor-optimized GPU BLAS kernels. In addition, the lack of a standard API or straightforward syntax for specifying contractions has can make a project unaccessible. Redesigning the syntax of these operations both eliminates transposition cost as well as divorces the algorithm from its domain-specific context.

A notable contribution was made by the general-purpose tensor operation library, TAL-SH. It is based on carefully optimized tensor transposition routines for popular parallel hardware,[8] and the source has become available since the original draft of this work was posted.

A few researchers have considered simply finding optimal methods for translating of tensor contractions to standard BLAS calls.[11, 12, 15] At a best, these translations still incur the cost of added tensor transpositions. In the worst case, the translation requires multiple level-1 BLAS operations with sparse data accesses that cannot be made cache-efficient and thereby slow execution times.

As an alternative strategy, the present work implements tensor contraction directly without the need for transposition. The contraction routines follow the general strategy pioneered by the Matrix algebra on GPU and multicore architectures (MAGMA) library,[6] but adds awareness for multi-index input and output tensors. The result is an identical computation for each tile of the output matrix, using modified read and write loops for accessing those tile locations.

Section 2 of this paper presents a low-level API comprised of only three basic operations, – **tscale**, **tadd**, and **tdot**. These cover all basic addition, multiplication, transposition, and contraction operations (Table 1). They are therefore a generalization of the core of BLAS levels 1-3 – excluding routines involving packed storage, symmetry, L_1 norms, or specialized rotation functions. Section 3 outlines the operation of the code generator which achieves high performance kernels by automatically generating implementation code for arbitrary thread block and output tile sizes.

A performance study is presented in Sec. 4. Using the same tile sizes as MAGMA without separate optimization or using texture mapping, the kernels corresponding to SGEMM (single precision general matrix multiply) are about 71% of the speed of the performance-tuned MAGMA library. Sig-

nificantly, these speeds are maintained for multi-index tensor products, where they hold a significant advantage over a naïve BLAS-based implementation strategy. It is shown that emulating those tensor products by a (faster) SGEMM sandwiched between tensor transpositions decreases overall performance by up to a factor of 4.

2. LOW-LEVEL CONTRACTION OPERATIONS

The two routines, **tadd** and **tdot**, cover all addition, multiplication, transposition, and contraction operations. The additional routine, **tscale**, is the equivalent of BLAS' **_SCAL**, and is included for completeness. The routine **tadd** is a fused tensor transpose plus add. Its inputs are two tensors, $A \in \mathcal{T}_{s_0, s_1, \dots, s_{n-1}}$ and $B \in \mathcal{T}_{t_0, t_1, \dots, t_{n-1}}$, of the same dimensionality (n), along with an index permutation, σ , mapping uniquely from indices of B to indices of A . The two tensors must have equivalent shapes under permutation, so $t_k = s_{\sigma(k)}$.

For compact notation, we define multi-indices as a complete tuple of indices, $I = (i_0, i_1, \dots, i_{n-1})$, $i_k \in \{0, 1, \dots, s_k - 1\}$, and permutations of tuples as $I = \sigma J$, where $i_{\sigma(k)} = j_k$. For numerical efficiency, two scaling factors are included in **tadd** inputs, and the result of the operation is stored in-place in the tensor A . The combined operation is logically,

$$\mathbf{tadd} \quad A[I] := \alpha A[I] + \beta B[\sigma^{-1}I] \quad (1)$$

The use of a permutation on the indices of B is necessary to remove the need for transpositions of input arguments. The input matrix, A is not transposed. Its index order carries through to the output. Necessary transposes can usually be folded into the eventual function consuming the output of '**tdot**'. Nevertheless, a matrix transpose as expressed by *numpy*'s **transpose**(A, σ^{-1}), can be written as **tadd**(0.0, Zero, 1.0, A , A .shape, σ).

Finally, the structure of '**tadd**' can be relaxed to allow both reductions along several indices of B or introduction of new indices into B . Both can be understood as broadcasting operations[20] involving an index of size 1 from one array and an index with an arbitrary size from the other. Note that inserting an index of size 1 is only a logical operation, and does not change the memory layout of the array.

Formally, we relax the restriction that $t_k = s_{\sigma(k)}$ to either $t_k = s_{\sigma(k)}$ or else one of: $t_k = 1$ or $s_{\sigma(k)} = 1$. The permutation still maps uniquely from indices of B to indices of A . All memory accesses to A and B are computed via strides. The stride is simply set to zero for any index with size 1. Explicitly, $A[I]$ is read from the memory location,

$$A + \sum_{k=0}^{n-1} i_k \text{stride}_k \quad (2)$$

where

$$\text{stride}_k = \begin{cases} 0, & s_k = 1 \\ \prod_{w=0}^{k-1} s_w, & \text{o.w.} \end{cases} \quad (3)$$

In the non-broadcast case, the stride for dimensions with size 1 is irrelevant, since the only allowed index is zero. In the broadcast case, the stride of zero ensures that all indices into that dimension appear as copies of the zero-index. For a reduction, such as $A := A + \beta \sum_j B_j$, an index is inserted

Figure 1: Role of index permutations in the tensor contraction. Permutations of A and B indices specify both the order of the contraction result, C , and an artificial ordering on the contracted indices. For comparison, a common emulation using GEMM sandwiched between tensor transpositions is shown below.

into A with size 1 and stride 0. Summation along j will accumulate into the same output location. For a broadcast, such as $A_{ij} = \alpha A_{ij} + \beta B_j$, a new first index is added to B with size 1 and stride 0.

Many other useful functions also follow the same basic pattern of `tadd`, replacing ‘+’ with any commutative operation on 2-arguments. The direct product just replaces ‘+’ with, ‘*’. Reductions can usefully replace ‘+’ with ‘min’ or ‘max’, etc., but require atomic updates when each thread makes its final output.

2.1 Tensor Contraction

The ‘`tdot`’ operation expresses a general tensor contraction. Tensor contractions are a special type of marginalization product function[18] where indices in $A \in \mathcal{T}_{s_0, \dots, s_{n_A-1}}$ (and respectively $B_{t_0, \dots, t_{n_B-1}}$) are divided into two disjoint sets: n “inner” and $n_A - n$ (resp. $n_B - n$) “outer” indices. The inner indices are to be summed over and must have sizes that match between A and B . The outer indices are not constrained, and are output to a third tensor, C . This output tensor therefore has dimension $n_C = n_A + n_B - 2n$.

We specify a generic tensor contraction using two permutations, σ_A and σ_B , mapping indices in A (resp. B), to indices in an expanded, logical tensor, $C' \in \mathcal{T}_{u_0, \dots, u_{n_C+n-1}}$. The complete tensor contraction is a sum over the last n indices of this expanded tensor,

$$\text{tdot } C[u_0, \dots, u_{n_C-1}] := \alpha \sum_{u_{n_C}, \dots, u_{n_C+n-1}} C'[u_0, \dots, u_{n_C+n-1}] + \beta C[u_0, \dots, u_{n_C-1}] \quad (4)$$

$$C'[u_0, \dots, u_{n_C+n-1}] = A[i_0, \dots, i_{n_A-1}] B[j_0, \dots, j_{n_B-1}] \quad (5)$$

The expanded tensor is illustrated in Fig. 1. The indices $\{u\}$ are such that $u_{\sigma_A(k)} = i_k$ and $u_{\sigma_B(l)} = j_l$. The outer indices of A (B) thus map uniquely to the left part of C' . The index, k , of tensor A (resp. l of B) is outer if $\sigma_A(k) < n_C$ ($\sigma_B(l) < n_C$), and contraction enforces $\sigma_A(k) \neq \sigma_B(l)$.

The inner indices go to the right in pairs, so k' is an inner index of A (resp. l' of B) if $\sigma_A(k') \geq n_C$ ($\sigma_B(l') \geq n_C$). Contraction enforces exactly one equivalent map, $m \iff (k', l')$, where $u_m = \sigma_A(k') = \sigma_B(l')$, for each $m \geq n_C$.

The `tdot` operation encompasses a large subset of BLAS 1-3. Table 1 lists several translations.

A straightforward implementation of tensor contraction is provided by `tensordot` in the *numpy* library with a fixed ordering of all outer indices.[20] First the tensors A and B are transposed to A' , B' , so that the inner indices of both are moved to the right-hand side in matching order. The outer indices are left in their input ordering. Because they are contiguous, all outer indices can now be addressed using a single, combined index, and similarly for the inner indices. The tensor product then reduces to a matrix product, $A'B'^T$. To emulate the present definition of contraction, this end result needs to be supplemented by a final transpose to re-order the output indices. The efficiency of this method is compared to the direct kernel versions in Sec. 4.

3. CODE GENERATION STRATEGY

The low-level kernel execution for `tdot` is essentially a GEMM, but has the added difficulty of dealing with arbitrary numbers of indices. The most simplistic implementation is to combine all n_C output indices of C into one omnibus dimension, and all n inner indices into another. Each iteration of a double-loop decodes the corresponding locations in A , B , and C . The timings for this implementation, using 1 CUDA thread per output location peak around 10 Gflops and are in most cases worse than the transpose-sandwiched version.

A much better performance is achieved by adopting the MAGMA library strategy of loop nesting optimized for the Fermi architecture.[6, 13] There, a $T_A \times T_B$ block of threads each compute $W_A \times W_B$ output locations to produce an $T_A W_A \times T_B W_B$ output tile of C for each thread block. Each thread sums over the whole contracted dimension, K . The outline of the procedure is as follows:[13]

1. Each of the $T_A \times T_B$ block of threads allocates and zeros a $W_A \times W_B$ accumulator, `acc` in thread-local, register space.
2. An outer loop over contracted indices works on N additions into `acc` at a time:
 - (a) All threads work together to pre-fetch a $T_A W_A \times N$ tile of A and a $T_B W_B \times N$ tile of B into shared memory.¹ Synchronization is required around this step.
 - (b) Threads work individually to accumulate N additions into `acc` using GER-like operations.
3. Each thread writes `acc` to the appropriate output locations in C .

¹ An additional optimization done by MAGMA and the present work is to have each thread pre-fetch the next tile of A and B as early as possible into thread-local register space. Then the shared memory is filled from those registers rather than directly from global device memory. In practice, the next copy starts just before each accumulation step. The assignment of parts of A and B to pre-fetch is unrelated to the thread’s output location.

$x \leftarrow \alpha x$	<code>_SCAL(N, α, X)</code>	<code>tscale(α, X, N)</code>
$y \leftarrow x$	<code>_COPY(N, X, Y)</code>	<code>tadd(0.0, Y, (N), 1.0, X, (N), (0))</code>
$y \leftarrow \alpha x + y$	<code>_AXPY(N, α, X, Y)</code>	<code>tadd(1.0, Y, (N), α, X, (N), (0))</code>
$r \leftarrow x^T y$	<code>_DOT(N, X, Y)</code>	<code>tdot(0.0, R, 1, 1.0, X, (1, N), (0, 1), Y, (N), (1))</code>
$y \leftarrow \alpha A x + \beta y$	<code>_GEMV('T', M, N, α, A, X, β, Y)</code>	<code>tdot(β, Y, 1, α, A, (N, M), (0, 1), X, (M), (1))</code>
$y \leftarrow \alpha A^T x + \beta y$	<code>_GEMV('N', M, N, α, A, X, β, Y)</code>	<code>tdot(β, Y, 1, α, A, (N, M), (1, 0), X, (N), (1))</code>
$A \leftarrow \alpha x y^T + A$	<code>_GER(M, N, α, Y, X, A)</code>	<code>tdot(1.0, A, 2, α, X, (N, 1), (0, 2), Y, (M, 1), (1, 2))</code>
$C \leftarrow \alpha A B + \beta C$	<code>_GEMM('N', 'N', M, N, K, α, B, A, β, C)</code>	<code>tdot(β, C, 2, α, A, (N, K), (0, 2), B, (K, M), (2, 1))</code>
$C \leftarrow \alpha A^T B + \beta C$	<code>_GEMM('N', 'T', M, N, K, α, B, A, β, C)</code>	<code>tdot(β, C, 2, α, A, (K, N), (2, 0), B, (K, M), (2, 1))</code>
$C \leftarrow \alpha A B^T + \beta C$	<code>_GEMM('T', 'N', M, N, K, α, B, A, β, C)</code>	<code>tdot(β, C, 2, α, A, (N, K), (0, 2), B, (M, K), (1, 2))</code>
$C \leftarrow \alpha A^T B^T + \beta C$	<code>_GEMM('T', 'T', M, N, K, α, B, A, β, C)</code>	<code>tdot(β, C, 2, α, A, (K, N), (2, 0), B, (M, K), (1, 2))</code>

Table 1: Translation from C, row-major shorthand notation to legacy BLAS and to `tadd` / `tdot` for arrays of dimension 1 and 2. All lower dimensions (IDX, etc.) are 1. Argument order for `tadd` is α , A, A shape, B, B shape, B permutation. Argument order for `tdot` is β , C, dimension of C, α , A, A shape, A permutation, B, B shape, B permutation. Note that `tdot` is symmetric to interchanging A and B.

This strategy decreases the cost of memory movement by relying on the hardware to combine reads from all threads into sequential accesses of A and B from CUDA global memory. Both copies from A and B as well as computation of the output tile are parallelized among the thread block.

We have adapted MAGMA’s tile-based strategy to `tdot` by defining the output tile to be composed of 2 omnibus dimensions – all output indices from A and all output indices from B. The procedure outline is the same, but access to tiles of A, B, and C is strided by mapping the single thread index to the tensor multi-index. The code generator works for fixed tensor dimensions and permutations, so that the index decoding steps are explicit instead of loops over arbitrary number of dimensions. Copies in the nested inner loops have known sizes, and are written out explicitly without requiring integer division operations.

Although the output tile is logically indexed by output dimensions of A and then B (as for the output of `tensordot` – Fig. 1), the actual output is done directly to the correct locations in C. It has a large practical advantage in that the physical memory is only accessed once.

We experimented with multiple strategies for organizing kernels. In the naming convention we adopted, each kernel name has 3 parts,

- The tile size of the logical output (both inner and outer indices) from the whole thread block – $n_C + n$ integers.
- The number of threads working on each output index – n_C integers, each dividing the output size.
- The permutations, σ_A and σ_B .

The cache size for the entire thread block can be computed from the logical output tile size, s , as

$$\prod_{k=0}^{n_A-1} s_{\sigma_A(k)} + \prod_{k=0}^{n_B-1} s_{\sigma_B(k)} \quad (6)$$

for prefetching,

$$+ \prod_{k=0}^{n_C-1} s_k \quad (7)$$

for accumulation, and

$$+ \prod_{\{k|\sigma_A(k) < n_C\}} s_{\sigma_A(k)} + \prod_{\{k|\sigma_B(k) < n_C\}} s_{\sigma_B(k)} \quad (8)$$

for optimized copying from shared memory during the accumulation loop.

This naming scheme is a compromise that simplifies the usage of the kernel, but does not specify the division of prefetches among processors. Even for a 2-index contraction of 4-dimensional tensors like that in Fig. 1, there are 6 logical output dimensions and 4 dimensions to choose the thread block size. Because the search space is large, the kernel parameters tested were based on MAGMA tile sizes or chosen heuristically by picking larger tile sizes for the most memory-contiguous dimensions.

4. PERFORMANCE RESULTS

This section presents timing results for single-precision `tdot` operations computed on an NVIDIA GTX980 (1.38 GHz, 4 GB memory) with NVIDIA proprietary kernel driver version 346.72 for Ubuntu server 14.10 compiled with `nvcc` 7.0.27. Reported timings are the fastest out of 10 trials and do not include CPU to GPU transfer times. This GPU is one of the first releases to use the new Maxwell redesign of the K20. Because it has been designed for consumer graphics, there are fewer double-precision floating point units per stream processor. MAGMA timings for DGEMM (double precision matrix multiply) on this GPU peaked around 150 Gflops, while those for SGEMM were significantly higher at 4000 Gflops. Because of this limitation, only single precision timings are reported below.

The GM204 chip used by the GTX980 schedules thread blocks nondeterministically to one of 16 streaming multiprocessors (SMs). All blocks scheduled to a given SM utilize that SM’s 96 kB block of shared memory and its 64 kB L1/texture cache for holding register values. Differing from the Kepler architecture, the register and shared memory spaces are separate. Within the SM, each thread block is further divided into warps of 32 threads each. Warps assigned to a scheduler can be either running an instruction in parallel (1 CUDA core per thread) or waiting for memory access. On the GM204, there are 4 warp schedulers per SM.

For an example, suppose a standard matrix multiplication is scheduled using a thread blocks containing 96 threads

each. Those blocks would be divided into 3 warps. If an SM undertook to execute 2 thread blocks at a time, it would be managing 6 warps - each of which would alternate between prefetching elements and doing additions from shared memory to register space before finally outputting to tiles of C .

Fig. 2 compares alternative methods for computing the 4-index tensor contraction illustrated in Fig. 1. The horizontal axis shows N , the size of each dimension for the case where all shapes are $\{N, N, N, N\}$ hypercubes. The upper panel shows throughput for the SGEMM kernel on matrices of size $N^2 \times N^2$. The right side of Fig. 1 shows how the central step of the contraction is essentially a matrix multiplication. The test kernel computes one output element of the matrix per thread, and has fairly constant performance around 10 Gflops. The optimized MAGMA 1.6.1 kernel and cuBLAS performance is significantly higher, peaking around 4500-5000 Gflops (very near the theoretical peak of 5234). MAGMA uses a 16×16 thread block to compute 36 output elements each - for an output tile size of 96×96 . cuBLAS shows variable performance, likely the result of a larger tile size.

Notably, Fig. 2 shows that **tdot** achieves performance on the same order of magnitude as state of the art optimized BLAS routines. Using the same tile sizes as MAGMA (16×16 thread block and 96×96 output tile) for the generated **tdot** kernel gives around 2500 Gflops compared to MAGMA's 3500. Because the basic outline of the kernel is identical between MAGMA and **tdot**, the lower performance may be due to minor differences in the layout and orderings of copies between global memory, shared memory, and registers.

The lower panel of Fig. 2 compares the overall performance for the full 4-index contraction of Fig. 1 using a **tdot** kernel created to directly compute the contraction against an emulation strategy using SGEMM sandwiched between transposes. The emulated timings are computed by adding the timings for transposing both inputs, performing SGEMM using the MAGMA library, and transposing the final output. Even though the matrix multiplication step scales as $O(N^6)$, while transposition scales as $O(N^4)$, the matrix transpositions severely limit the overall throughput. It should be noted that tensor sizes larger than $N = 128$ could not be tested as they exceed the 4GB memory limit of the device.

Oscillations in throughput in Fig. 2 are due to many cases tested where the tile size does not divide the tensor size. For high-order tensors, these become the common case. Because padding space grows exponentially in the number of dimensions, padding is not feasible without resorting to the transpose strategy. Comparison between CUBLAS and MAGMA is not direct, since MAGMA was not specifically tuned for the current, Maxwell architecture.

To test a tensor contraction case with some low-dimensional indices, Fig. 3 presents performance results for the 2-index contraction,

$$C_{ij\alpha} = \sum_{k\beta} A_{ik\alpha\beta} B_{kj\beta}, \quad (9)$$

where indices $\alpha, \beta \in \{0, 1\}$ are only size two. The variable N labels the size of the i (and also the j, k) dimension. This is the matrix representation of complex multiplication when

$$A_{ik} = \begin{bmatrix} a_{ik}, -a'_{ik} \\ a'_{ik}, a_{ik} \end{bmatrix}. \quad (10)$$

Figure 2: Performance comparison between MAGMA and generated contraction kernel, **tdot.** The upper panel compares the fastest versions of the SGEMM kernel, which happened to be NN for MAGMA and NT for **tdot**. The lower panel shows that including transposes greatly decreases overall throughput for emulating **tdot**, while a direct 4-index **tdot** kernel performs at about half the speed of **tdot** for SGEMM. The designations NN and NT refer to the index ordering of the input matrices, as in Table 1.

Figure 3: Performance of an $N \times N$ complex matrix multiplication with CUBLAS/MAGMA compared against an $[N \times N \times 4] \cdot [N \times N \times 2]$ contraction with `tdot`. In this comparison, the number of floating point operations is the same, but the contraction requires twice as many reads. The difference from the top panel of Fig. 2 clearly illustrates the impact of memory accesses.

The tile-size used was identical to the MAGMA kernel, which seemed to give the best performance. Despite requiring twice as many loads from A , the performance of `tdot` is of the same order of magnitude as straightforward complex matrix multiplication. This also illustrates a case where tensors can play an important role, since higher-dimensional Clifford algebras (e.g. quaternions) share the same contraction structure, but with more α indices.

Fig. 4 provides further detail on the time spent during tensor transposition *vs* computing matrix multiplication. The transpositions occupy a significant fraction of time for all sizes. In fact, for most reasonable tensor sizes, the transposes occupy more than 60% of the time. The requirement for separate transpositions explains the low overall throughput for emulating a direct tensor contraction.

5. CONCLUSIONS AND FUTURE WORK

We have shown that computing tensor transpositions can be a significant bottleneck to general tensor algebra, both in terms of performance and memory requirements. Legacy BLAS routines do not provide a means to cope with this issue. Instead, we propose a simple and general API for combining transpositions directly into tensor contractions and summations. This provides a means to hide the cost of tensor transpositions inside the data access already performed during the normal course of tensor multiplication. We have shown a preliminary implementation of this API that provides speed comparable to high-performance BLAS.

Many factors influence the efficiency of `tdot` kernels. These include both the shape of the tensors A , B , and C relative to the thread and work block shape as well as the index permutations applied to A and B . In all cases, it is reasonable to expect the largest performance improvements when A , B , and C are all accessed sequentially within each tile.

Tensors present unique challenges in that high-dimensional tensors have many indices with small shapes. These make padding unfeasible in a tensor setting, since the size required

Figure 4: Breakdown of time spent in emulated `tdot` for Fig. 1. Host $\leftrightarrow$ device memory transfer time is not included. SGEMM timings are copied from MAGMA’s timings on an $N^2 \times N^2$ matrix. Transpose timings (performed on a 2.5 GHz Intel Core i7 processor with 1600 MHz DDR3 RAM) count only the transposes that would be done during `tensor_dot` within the *numpy* library. Even for large tensors, a majority of the time is spent in CPU transpose.

grows exponentially with the tensor dimensions. However, if a few small dimensions appear in the minor (sequential) indices of A , B , or C , then a custom kernel can be written to access these inputs sequentially. The ordering of the major indices of the tensors becomes less important as tensor sizes grow.

As expressed by Ref. [6], cases with a few small dimensions are precisely where code generation is most useful. They found large numbers of kernels with smaller tile sizes that performed nearly as well as those with larger tiles because of their ability to combine sequential access patterns with loop nesting working cache-sized-at-a-time.

Future work should continue to investigate more efficient ways of generating and choosing templates, as well as more high-level, science-friendly interfaces to linear operations. We did not have the opportunity to investigate methods to optimize host to device memory transfer – which could accomplish a partial transposition, or to explore non-square matrices. Developing an accurate cost model for device performance would help by allowing a fast scan of the kernel parameter search space. Fewer errors and greater reproducibility can also be gained by providing a high-level language to compose chains of `tdot` and `tadd` operations on multiple input and output tensors. A separate paper describes the SLACK library’s implementation of a DAG scheduler for tensor operations based on a $\LaTeX$ equation input syntax.

6. ACKNOWLEDGMENTS

This work was supported by the USF Research Foundation, NSF MRI CHE-1531590, and NSF XSeed Resource Allocation TG-ASC130043. We thank Jeff Hammond for

lively discussions leading to this work.

7. SOFTWARE

Reference and optimized implementations of the SLACK project are made available via (github.com/frobnitzem/slack). All code will be released under terms of the GNU GPL.

8. REFERENCES

- [1] E. Aprà, K. Kowalski, J. R. Hammond, and M. Klemm. *NWChem: Quantum Chemistry Simulations at Scale*, pages 287–306. Morgan Kaufmann, 2015.
- [2] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, P. S. J. Ramanujam, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proc. IEEE*, 93(2), 2005.
- [3] A. E. DePrince, M. R. Kennedy, B. G. Sumpter, and C. D. Sherrill. Density-fitted singles and doubles coupled cluster on graphics processing units. *Molecular Phys.*, 112(5–6):844–852, 2013.
- [4] J. Hammond. Automatically tuned libraries for native-dimension tensor transpose algorithms. 2008. <https://github.com/jeffhammond/spaghetty>.
- [5] E. G. Hohenstein, R. M. Parrish, C. D. Sherrill, and T. J. Martínez. Communication: Tensor hypercontraction. iii. least-squares tensor hypercontraction for the determination of correlated wavefunctions. *J. Chem. Phys.*, 137(22):221101, 2012.
- [6] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMMs for Fermi. *Proc. ACM/IEEE Conference on Supercomputing (SC '11)*, 2011.
- [7] C. L. Lawson, R. J. Hanson, R. J. Kincaid, and F. T. Krogh. Basic linear algebra sub-programs for Fortran usage. *ACM Trans. Math. Software*, 5:308–323, 1979.
- [8] D. I. Lyakh. An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. *Comput. Phys. Commun.*, 189:84–91, 2015.
- [9] W. Ma, S. Krishnamoorthy, O. Villa, and K. Kowalski. GPU-based implementations of the noniterative regularized-CCSD(T) corrections: applications to strongly correlated systems. *J. Chem. Theory Comput.*, 7(5):1316–1327, 2011.
- [10] W. Ma, S. Krishnamoorthy, O. Villay, and K. Kowalski. Acceleration of streamed tensor contraction expressions on gpgpu-based clusters. 2010 IEEE International Conference on Cluster Computing (CLUSTER), pages 207–216, 2010.
- [11] E. D. Napoli, D. Fabregat-Traver, G. Quintana-Ortí, and P. Bientinesi. Towards an efficient use of the BLAS library for multilinear tensor contractions. Aachen Institute for Advanced Study in Computational Engineering Science, 2013. technical report.
- [12] E. D. Napoli, D. Fabregat-Traver, G. Quintana-Ortí, and P. Bientinesi. Towards an efficient use of the blas library for multilinear tensor contractions. *Appl. Math. Comput.*, 235:454–468, 2014.
- [13] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *Int. J. High Perf. Comput. Applic.*, 24(4):511–515, 2010.
- [14] T. Nelson, A. Rivera, M. Hall, P. D. Hovland, E. Jessup, B. Norris, and P. Balaprakash. Generating efficient tensor contractions for GPUs. In *Proc. 44th International Conference on Parallel Processing (ICPP)*, 2015.
- [15] E. Peise, D. Fabregat-Traver, and P. Bientinesi. On the performance prediction of BLAS-based tensor contractions. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, volume 8966 of *Lecture Notes in Computer Science*, pages 193–212. Springer, 2015.
- [16] A. Y. Rivera. *Using autotuning for accelerating tensor contraction on graphics processing units (GPUs)*. Univ. Utah, 2014. Master’s Thesis.
- [17] H. Shan, S. Williams, W. de Jong, and L. Oliker. Thread-level parallelization and optimization of NWChem for the Intel MIC architecture. PMAM ’15 Proc. Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, pages 58–67. ACM New York, NY, USA, 2015.
- [18] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on gpus through software-managed cache. *ACM International Conference on Supercomputing*, pages 309–318, 2008.
- [19] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. *IPDPS, Parallel and Distributed Processing Symposium*, pages 813–824, 2013.
- [20] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13:22–30, 2011.