Tensor Contractions with Extended BLAS Kernels on CPU and GPU

Yang Shi *, U. N. Niranjan †, Animashree Anandkumar *
* EECS Department, † ICS Department
University of California, Irvine
Irvine, USA
Email: {shiy4,un.niranjan,a.anandkumar}@uci.edu

Cris Cecka
NVIDIA Research
Santa Clara, USA
Email: criscecka@gmail.com

Abstract—Tensor contractions constitute a key computational ingredient of numerical multi-linear algebra. However, as the order and dimension of tensors grow, the time and space complexities of tensor-based computations grow quickly. In this paper, we propose and evaluate new BLAS-like primitives that are capable of performing a wide range of tensor contractions on CPU and GPU efficiently. We begin by focusing on single-index contractions involving all the possible configurations of second-order and third-order tensors. Then, we discuss extensions to more general cases.

Existing approaches for tensor contractions spend large amounts of time restructuring the data which typically involves explicit copy and transpose operations. In this work, we summarize existing approaches and present library-based approaches that avoid memory movement. Through systematic benchmarking, we demonstrate that our approach can achieve 10x speedup on a K40c GPU and 2x speedup on dual-socket Haswell-EP CPUs, using MKL and CuBLAS respectively, for small and moderate tensor sizes. This is relevant in many machine learning applications such as deep learning, where tensor sizes tend to be small, but require numerous tensor contraction operations to be performed successively. Concretely, we implement a Tucker decomposition and show that using our kernels yields at least an order of magnitude speedup as compared to state-of-the-art libraries.

Keywords—Parallelism; BLAS; GPU; Tensor;

I. INTRODUCTION AND SCOPE

Multilinear algebraic computations, are ubiquitous in multiple scientific domains such as machine learning and modern data science [4], quantum chemistry and physics [14], signal and image processing [9], chemometrics [7], and biochemistry [13]. The study of tensor computations has a long and diverse history, as early as in the work by Hitchcock [11]. The domains and references provided herein are by no means exhaustive but merely a small representative sample of the various flavors in which tensor computations are used in science. Tensors are multi-way arrays which can be viewed as a generalization of matrices to allow multimodality in data. Tensor contractions play a central role in a variety of algorithms and applications; for a motivating example, see Section II-C. However, non-trivial performance bottlenecks in several application areas are encountered due to the high space and time complexities associated with tensor computations. In this paper, motivated by the recent increased interest from machine learning and deep learning, we propose and study library-based communication avoiding approaches for performing tensor contractions.

Conventional approaches for computing general tensor contractions rely on matricization, the logical or explicit restructuring of the data so that the computation can be performed with a sequence of Basic Linear Algebra Sub-routine (BLAS) library calls. The BLAS routines provide efficient and portable implementations of linear algebra primitives, with many fast implementations existing across many architectures [8].

To this point, the GEneral Matrix Multiply (GEMM) primitive specified within the BLAS library is possibly the most optimized and widely used routine in scientific computing. Noting that the basic theoretical computational and communication complexities of most tensor contractions is equivalent to that of GEMM, these computations should scale equally well. However, we find that existing tensor libraries such as the TENSOR TOOLBOX and CYCLOPS perform explicit data transposition to compute almost all tensor contractions and the cost of data restructuring often dominates the cost of the actual computation. Other approaches have previously proposed intrusive compiler and static analysis solutions, whereas we provide a much simpler library-based solution [16], [17].

Findings and contributions: We introduce a new BLAS primitive, known as STRIDEBATCHEDEMM, that allows the majority of tensor contractions to be computed without any explicit memory motion. We detail the so-called exceptional cases that cannot be evaluated with STRIDEBATCHEDEMM and demonstrate that an efficient solution exists with another small extension to the primitive.

We demonstrate performance improvement using our approach on both CPU and GPU in direct benchmarks in addition to an application study. The Tucker decomposition is an important tensor application in machine learning wherein the advantage of our strategy compared to existing libraries is clear.
Finally, the value of this approach and its applications are being recognized by NVIDIA. As of this writing, the proposed interface exists in the CuBLAS 8.0 Release Candidate and is likely to appear the official release later this summer.

II. BACKGROUND

A. Related Work

Peise et al. [22] extended results from Napoli et al. [19] in mapping tensor contractions to sequences of BLAS routines and modeling the performance of these mappings. In this work, they systematically enumerate and benchmark combinations of possible BLAS kernels one could use to compute a given tensor contraction to conclude that the best performing algorithms involve the GEMM kernel. Some evaluation strategies are neglected to be considered, such as flattening or developing new, generic linear algebraic subroutines that could yield improved performance.

Li et al. [16] also recognizes the cost of explicit copies and proposes evaluation strategies exactly comparable to the flattening and batching strategies addressed in this paper. Their discussion of loop modes and component modes map to our discussion of batch modes and GEMM modes. However, Li et al do not discuss strategies beyond tensor-times-matrix multiply. Furthermore, they only consider component times-matrix contractions of the form $Y_{ij} \cdots i_{n-1} j \cdots i_1 = \sum_{j} X_{i_1 \cdots i_{n-1} j \cdots i_{n}} U_{j i_n}$, which avoids the more complicated cases in this paper. Abdelfattah et al. [3] presents a framework using batched GEMM for tensor contractions on GPUs. However, they focus on optimizing only limited number of tensor contraction kernels on extreme small size tensors. Other works in [1] [20] improve the tensor computation performance by doing loop reorganization and fusion.

The STRIDED BATCHED GEMM interface proposed in this paper has previously been mentioned by Jhurani et al. [12] as a low-overhead interface for multiple small matrices on NVIDIA GPUs. Jhurani proposes the same interface for CuBLAS that we propose in this paper and focuses on implementation concerns. In this work, we treat STRIDED BATCHED GEMM as an available primitive, benchmark evaluation strategies that utilize it, and examine how it may be further extended for use in multi-linear algebra.

The BLAS-like Library Instantiation Software (BLIS) framework [26] offers GEMMs which support non-unit strides in both the row and column dimensions, which are attractive solutions to some of the problems in this paper. However, performance is expected to suffer due to decreases in cache line utilization, and SIMD opportunities.

Recent improvements in parallel and distributed computing systems have made complex tensor computation feasible. TensorFlow [2] can handle multi-linear algebra operations and it is primarily a data-flow and task-scheduling framework for machine learning.

B. Notation

We denote tensors by uppercase letters, indices by lowercase letters and index lists by calligraphic letters. We assume all indexing is zero-based. $\mathbb{R}$ denotes the set of real numbers.

The order of a tensor is the number of modes it admits. A scalar is a zeroth-order tensor, a vector is a first-order tensor, a matrix (say $A_{mn}$) is a second-order tensor with the rows (indexed by $m$) being the first mode and columns (indexed by $n$) being the second mode, and a three-way array (say $A_{mnp}$) is a third-order tensor with the first, second and third modes indexed by $m$, $n$, and $p$, respectively. Note that we use the term index to name a mode and iterate through the elements in that mode.

The dimension of the $i^{th}$ mode, denoted $\text{dim}(i)$, is the number of elements it contains. The dimension of a mode of a tensor is denoted by the bold lowercase letter of the respective index; for example, the third-order tensor $A_{mnp}$ has dimension $\text{dim}(0) \times \text{dim}(1) \times \text{dim}(2)$ or $m \times n \times p$ where the first mode (indexed by $m$) takes values $0, \ldots, m−1$, the second mode (indexed by $n$) takes values $0, \ldots, n−1$, the third mode (indexed by $p$) takes values $0, \ldots, p−1$.

We follow Einstein summation convention to represent tensor contractions. A general tensor contraction is written as

$$C_{C} = \alpha A_{A} B_{B} + \beta C_{C}$$

(1)

where $A, B, C$ are ordered sequences of indices such that $C \equiv (A \cup B) \setminus \{A \cap B\}$. The indices in $A \cap B$ are called contracted indices. The indices in $C$ are called free indices.

C. An Important Practical Application

In unsupervised learning, tensor decomposition [4] is gaining a lot of attention and is the crux of model estimation via the method of moments. A variety of problems such as topic model estimation, Gaussian mixtures model estimation, and social network learning can be provably, consistently and efficiently solved via the tensor decomposition techniques under certain mild assumptions.

The basic building blocks of these algorithms involve tensor contractions. Two frequently used tensor decomposition methods are the CP decomposition [10] and the Tucker decomposition [25]. In [27], the authors use the Tucker decomposition to extract new representations of the face images despite different expressions or camera viewpoints. To illustrate the fundamental importance of tensor contractions, we will pick one of the most common tensor decomposition algorithms, namely the higher-order orthogonal iteration (HOOI) [15] for asymmetric Tucker decomposition, and use it as a case-study. In the Einstein notation, the factorization of a third-order tensor $T \in \mathbb{R}^{m \times n \times p}$ is given by $T_{mnp} = G_{ijk} A_{mi} B_{nj} C_{pk}$, where $G \in \mathbb{R}^{i \times j \times k}$ is the core tensor, $A \in \mathbb{R}^{m \times i}$, $B \in \mathbb{R}^{n \times j}$, $C \in \mathbb{R}^{p \times k}$. From Kolda et al [24], we summarize the algorithm for the third-order tensor case in Algorithm 1. Following their notation, $T_{(r)}$ denotes the mode-$r$ unfolding of tensor $T$. For further technical details, we refer the reader to Kolda et al [24].
Algorithm 1 Tucker decomposition algorithm.

Require: Tensor $T \in \mathbb{R}^{m \times n \times p}$, core tensor size i, j, k, number of iterations $T$.

Ensure: Factors $A^T_i$, $B^T_j$, $C^T_k$ and core tensor $G$.

1: Set $t = 0$;
2: Initialize $A^{0}_i$ leading left singular vector of $T_{(1)}$.
3: $B^{0}_j$ leading left singular vector of $T_{(2)}$.
4: $C^{0}_k$ leading left singular vector of $T_{(3)}$.
5: while $t < T$ do
6: $Y_{mjk} = T_{mnp} B^T_{nj} C^T_{pk}$
7: $A^{t+1}_i$ leading left singular vector of $Y_{(1)}$.
8: $B^{t+1}_j$ leading left singular vector of $Y_{(2)}$.
9: $C^{t+1}_k$ leading left singular vector of $Y_{(3)}$.
10: end while
11: $G_{ijk} = T_{mnp} A^T_{mi} B^T_{nj} C^T_{pk}$

D. Conventional Tensor Contraction

The conventional approach for tensor contraction is to matricize the tensors via transpositions and copies. Libraries such as Basic Tensor Algebra Subroutines (BTAS) [18], MATLAB Tensor Toolbox [6], [5], and Cyclops Tensor Framework [23] all perform some version of matricization, which is typically performed in four steps:

1) Consider a general tensor contraction of the form (1).
2) Permute tensors $A$, $B$, and $C$ into the form
$$C_{I,J} = \alpha A_{IK} B_{K,I} + \beta C_{I,J}$$
3) Evaluate (2) using one of four BLAS kernels:
$$\begin{cases} \text{DOT} & |\mathcal{K}| = |\mathcal{A}| \text{ and } |\mathcal{K}| = |\mathcal{B}| \\ \text{GER} & |\mathcal{K}| = 0 \\ \text{GEMV} & |\mathcal{K}| = |\mathcal{A}| \text{ xor } |\mathcal{K}| = |\mathcal{B}| \\ \text{GEMM} & \text{else} \end{cases}$$
4) Permute the result, $C_{I,J}$, into the desired output, $C_C$.

This approach to tensor contractions is completely general – it works for any two tensors of arbitrary order and any number of contraction indices. However, for even the simplest contractions, the cost of explicitly permuting the tensor data typically outweigh the cost of the computation to be performed. See Section III-A for examples.

III. APPROACH

In this section, we present library-based evaluation strategies for performing general tensor contractions in-place – without explicit copies and/or transpositions.

Figure 1: The fraction of time spent in copies/transpositions when computing the contraction $C_{mnp} = A_{mk} B_{pkn}$ using the conventional approach. Lines are shown with 1, 2, 3, and 6 total transpositions performed on either the input or output. (Left) CPU. (Right) GPU.

A. Motivating Observations

Case study 1 : Consider $C_{mnp} = A_{mk} B_{pkn}$. The conventional approach presented in Section II-D results in an evaluation wherein one switches, by means of explicit copy operations, modes $n$ and $k$ in $B$ to produce $C_{mnp} = A_{mk} B_{pkn}$, which is now of the form (2) and can be evaluated directly with a GEMM. Alternatively, we observe that we may perform the computation without explicit copy by launching $p$ individual GEMMS.

Case study 2 : Consider $C_{mnp} = A_{km} B_{pkn}$. The conventional approach presented in Section II-D results in an evaluation wherein we may require more than one transposition. For concreteness, we analyzed how BTAS performs this contraction. We observed that BTAS uses four explicit transpositions that results in the following algorithm:

1) Permute $A_{km}$ to $A_{mk}$.
2) Permute $B_{pkn}$ to $B_{kpn}$.
3) Permute $C_{mnp}$ to $C_{mpn}$.
4) Compute $C_{mpn} = \alpha A_{mk} B_{pkn} + \beta C_{mpn}$ with GEMM.
5) Permute $C_{mpn}$ to $C_{mnp}$.

Similarly, in the MATLAB Tensor Toolbox, the main idea is to reshape all tensors to matrices. For instance, in Case 2.4 in Table II, it reshapes $A_{km}$ to $A_{mk}$ and reshapes tensor $B_{pkn}$ to matrix $B_{k(pn)}$ with the first dimension as $k$ and the second dimension as $p \times n$. Cyclops also uses index reordering methods for fully dense tensors. The reordering is avoided only in the more restrictive case of high-dimensional symmetric tensors.

We note that some of the steps in the above approach can certainly be avoided with an improved algorithm that still implements the conventional approach. For example, Step 1 can be avoided by using a GEMM that implicitly transposes the first matrix via a CblasTrans parameter or equivalent in Step 4. Another optimization would be to avoid Step 3 altogether when $\beta = 0$. Other approaches require even fewer transposition steps. Ultimately, observe that we may perform the computation without explicit copy by performing $p$ individual GEMMS.
In Figure 1, we measure the cost of these explicit transpose operations in a representative tensor contraction on CPU and GPU. On the CPU we use MKL’s mkl_somatcopy and cblas_sgemm, and on the GPU we use CuBLAS’s cublasSgeam and cublasSgemm to perform each required matrix transposition and GEMM respectively. Note that transposition primitives are not specified in BLAS, but are vendor-specific BLAS-like extensions provided to perform common transpose operations. For this reason, these optimized functions are not typically used in tensor libraries with, instead, custom transposition implementations taking their place. These custom implementations are likely not as optimized as the vendor implementations.

As we can see from Figure 1, on the CPU, almost 40% of the time is used in copy and transpose, even when only a single mode transposition is performed. Clearly, with more transpose operations, the fraction is higher, requiring 60-80% of the total time. This correlates well with data presented in [16] where it is reported that Tensor Toolbox takes approximately 70% of the total time performing copies and transpositions in one algorithm. By avoiding these transpositions we may obtain 10x speedup on the GPU for small tensors with $n \lesssim 100$, and more than 2x speedup on the CPU for almost all $n$.

Although the fraction of time spent in transpose will asymptotically approach zero as $n$ grows in both cases, the high bandwidth of the GPU allows the computation to dominate the communication much more quickly. Indeed, the reported maximum bandwidth of the K40c GPU is 288GB/sec and the dual-socket Xeon E5-2630 v3 CPU achieves 118GB/sec.

Additionally, that the gap between computational performance and communication performance continues to increase, so the cost of transposition is likely to increase in the future. Even now, especially for small tensor sizes, it is clear that the cost of performing explicit copies and transpositions is significant and should be avoided.

**B. Extended Notation**

We would like to express evaluation strategies for tensor contractions succinctly, so we introduce additional notation.

In this paper, tensors are assumed to be stored in the column-major format. In other words, the $i^{th}$ mode has a memory stride – termed “leading dimension” in BLAS – denoted $ld<i>$ with $ld<0> = 1$. Using this notation, $A_{mnp}$ is stored as $A[m + n \times ld<1> + p \times ld<2>]$. Note that the common packed-storage case is obtained when, for all $i$, we have $ld<i> = \prod_{0 \leq c < i} \dim<k>$.

We now formalize three operations that are used in tensor contraction evaluations.

1) Batching: $[i]$ denotes that mode $i$ is batched. A batched mode is considered fixed.

2) Flattening: $(ij)$ denotes that modes $i$ and $j$ are flattened, i.e., modes $i$ and $j$ are now considered together as a single mode. The combined mode $h = (ij)$ is considered free.

3) Transpose: $A^T_{mn}$ denotes a matrix transpose. Transposes may only be applied to tensors with exactly two free modes.

The purpose of these notations is that they map directly to looped BLAS calls and the appropriate evaluation can often be read directly from the notated expression. Next, we review some rules that the above notation must follow in order to obtain a well-formed evaluation expression.

1) A batched mode $[i]$ cannot be the first mode of any matrix term. That is, $A_{[m]nk}$ is not allowed. Batching in the first mode would cause the $m$ resulting logical $n \times k$ matrices to be strided in both rows and columns and, therefore, cannot be used as a matrix in any BLAS routine.

2) A flattening $(ij)$ requires that $ld<j> = ld<i> \cdot \dim<i>$. Unfortunately, the notation alone is therefore not sufficient to determine which modes may be flattened; it is contingent on the representation as well. In the common packed-storage case, however, this flattening condition is always true.

3) If a flattening operation occurs on the right side, it must occur on the left side with the same modes in the same order. For example, $C_{m(np)} = A_{mk}B_{k(np)}$ is not allowed.

4) Standard transposition rules apply: $C^T_{nm} = A^T_{mk}B_{kn}$ implies $C_{nm} = B^T_{nk}A^T_{mk}$. However, modes may not be swapped under transposition. For example, $A^T_{mk}$ can not be replaced with $A_{km}$.

This notation allows us to quickly read off the intended extended BLAS evaluation expression for arbitrary tensor contractions. See Table I for examples.

**C. BatchedGEMM**

Instead of relying on explicit mode transpositions, Peise et al [21], [22] considered mapping tensor contractions to BLAS primitives directly – enumerating all possible BLAS primitives that could be used and their nesting within loops.

![Figure 2: The arithmetic intensity of computing $n$ GEMMs of size $n \times n$ versus the achieved performance on a K40c GPU and 16 cores (32 threads) of a dual socket CPU.](image-url)
Of course, the evaluation strategies that relied on level-3 BLAS primitives (GEMM) rather than level-2 primitives (GEMV, GER) were much more efficient. This often resulted in the need for many small GEMMs to be performed, which usually does not achieve ideal performance.

The need to compute many small GEMMs has not gone unnoticed by the leading implementations of BLAS. NVIDIA supplied the capability to multiply pairs of many small matrices in CUBLAS v4.1 [CUDA Toolkit v4.1] via the function cublasSgemmBatched. Similarly, as of MKL 11.3.1, cblas_Xgemm_batch is available with a similar interface and is also specifically optimized for small matrix sizes.

In Figure 2, we plot the achieved performance on CPU and GPU of these BatchedGemm functions by evaluating many GEMMs of size n x n using each strategy with MKL 11.3.1 and CUBLAS 7.5. Note that there are much higher performance in both cases when n is small. When n is large, there is clearly room for optimization in cUBLAS and CUBLAS.

Both of these interfaces are based on pointers to matrix pointers, which often require allocation and/or precomputation at the point-of-call. This makes them awkward to use in the context of tensor contractions where the strides between matrices are regular and the generality provided by these interfaces goes unused.

### D. StridedBatchedGemm

Building on the BatchedGemm extensions to BLAS, we propose StridedBatchedGemm (Listing 1) which offers a simplified interface for the constant-strided BatchedGemm and more optimizations opportunities for implementors. The interface and reference implementation of StridedBatchedGemm is provided in Listing 1. The lda, ldb, ldc parameters are the standard “leading dimension” parameters that appear in level-3 BLAS primitives and denote to the stride between columns of the matrix. We refer to the new loc, lob, loc parameters as the “leading order” parameters and denote the stride between matrices of the batch.

There are a number of advantages to a StridedBatchedGemm primitive. First, StridedBatchedGemm is actually more restrictive than the BatchedGemm that has already appeared in MKL and CUBLAS, but we argue that a BatchedGemm with a constant stride between matrices is a common enough case to consider specializing for. By providing this interface, the common case with constant strides between matrices is not forced to perform allocations or precomputations as it currently must perform in order to use BatchedGemm. Additionally, these extra restrictions provide additional knowledge of the memory layout of the computation and offers additional optimizations opportunities in SIMDization, prefetching, and tiling. In other words, the “batch-loop” in StridedBatchedGemm now directly participates in the polyhedral computation as an affine for-loop.

With the pointer-interface in BatchedGemm, the “batch-loop” cannot fully participate in a polyhedral model of the computation and is certainly not a candidate for vectorization or cache blocking.

In Table II, we have enumerated all unique single-mode contractions between a second-order and third-order tensor using the notation from Section III-B. All but 8 contractions can be computed with only a single call to StridedBatchedGemm.

### E. Exceptional Cases

The eight exceptional cases in Table II – Cases 3.4, 3.6, 4.4, 4.6, 5.4, 5.6, 6.4, and 6.6 – occur when batching forces the evaluation to either be a BatchedGemm or violate the no-first-mode rule.

This can be resolved by making an extension to the operation parameters allowed for BatchedGemm. Typically, the available operation parameters are “normal”, “transpose”, “conjugate”, and “Hermitian”. To account for the exceptional cases, “extended X” could be added to allow violations of the no-first-mode rule and consider all three modes involved in the batching simultaneously.

For example, Case 3.6 and 6.4 could then be written

\[
C_{mn[p]} = B[p, mk] A_{nk}^\top
\]

and evaluated via

\[
sb\_gemm(OP\_EX\_N, OP\_T, \text{sb\_gemm}(OP\_EX\_T, OP\_N, M, N, K, I, J, B, ldb<1>, ldb<2>, A, lda<1>, 0, 0, C, ldc<1>, ldc<2>, F));
\]

<table>
<thead>
<tr>
<th>Contraction</th>
<th>BLAS Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_{m(np)} = A_{mk} B_{k(np)} )</td>
<td>GEMM (’N’, ’N’, m, np, k, 1, A, lda&lt;1&gt;, B, ldb&lt;1&gt;, 0, C, ldc&lt;1&gt;)</td>
</tr>
<tr>
<td>( C_{(m)n} = B_{(m)k} A_{pk}^\top )</td>
<td>GEMM (’N’, ’T’, mn, p, k, 1, B, ldb&lt;1&gt; · ldb&lt;2&gt;, A, lda&lt;1&gt;, 0, C, ldc&lt;1&gt; · ldc&lt;2&gt;)</td>
</tr>
<tr>
<td>( C_{m[n]} = A_{mk} B_{k(np)} ) for ( n \in [0, n) )</td>
<td>GEMM (’N’, ’N’, m, p, k, 1, A, lda&lt;1&gt;, B + n · ldb&lt;1&gt;, ldb&lt;2&gt;, 0, C + n · ldc&lt;1&gt;, ldc&lt;2&gt;)</td>
</tr>
<tr>
<td>( C_{m[p]} = B_{k[p]m} A_{kn} ) for ( p \in [0, p) )</td>
<td>GEMM (’T’, ’N’, m, n, k, 1, B + p · ldb&lt;1&gt;, 1 · ldb&lt;2&gt;, A, lda&lt;1&gt;, 0, C + p · ldc&lt;2&gt;, 1 · ldc&lt;1&gt;)</td>
</tr>
<tr>
<td>( C_{[n]} = B_{p[k]n} A_{kn} ) for ( n \in [0, n) )</td>
<td>GEMV (’N’, p, k, 1, B, ldb&lt;1&gt;, A + n · lda&lt;1&gt;, 1, 0, C + n · ldc&lt;1&gt;)</td>
</tr>
</tbody>
</table>
When the extended operation is passed, it is known that the corresponding element of the input which always has leading dimension 1. Thus, the leading order parameter to leading order parameters.

The implementation of a computation like this is expected to perform a “3D” tiling of B into cache in order to efficiently contract with the standard 2D cache tiling of A.

F. Generalization

In this section, we explain the generality of our approach and how it can be easily applied and extended to single-mode contractions involving tensors of arbitrary order.

Consider an arbitrary single-mode tensor contraction of the form (1). It is straightforward by simple counting that the number of unique contractions is \(|A| + |B| - 2| \cdot |A| \cdot |B| \). We note that Table II is obtained with |A| = 2 and |B| = 3. Of these contractions, all of them may be performed without explicit mode transpositions by nesting the BatchedGemm operations.

We observe that some single-mode contractions of two tensors of arbitrary order can be evaluated by batching on different modes with the BatchedGemm operations. For example, consider \( C_{mn[p]} = A_{mk[p]} B_{nk[q]} \) wherein we can batch in either p and q. We prefer to choose the mode with the larger dimension for the BatchedGemm batching loop over the other (nested batching).

The nested-batching strategy in Listing 2 is general and extends to any two tensors of any order. Algorithms and heuristics for choosing the looped, batched, and GEMM-ed modes are provided in Section IV-D.

Table II: List of 36 possible single mode contraction operations between a second-order tensor and a third-order tensor and possible mappings to Level-3 BLAS routines. Note that 8 cases may be performed with GEMM, 28 cases may be performed with StridedBatchedGemm, and 8 cases remain exceptional.

Listing 1: Interface and reference implementation of BLAS-like strided batched GEMM.

```
1 // C_p = alpha*opA(A_p)*opB(B_p) + beta*C_p
2 sb_gemm(op_type A_p, op_type opB, C_p);
3 int m, n, k,
4 T alpha,
5 const T* A_p, int lda, int loa,
6 const T* B, int ldb, int lob,
7 T beta,
8 T* C, int ldc, int loc,
9 int batch_size)
10 {
11 // EXPOSURE ONLY
12 for (int p = 0; p < batch_size; ++p)
13     gemm(opA, opB,
14         m, n, k,
15         alpha,
16         A_p + p*lda, lda,
17         B + p*ldb, ldb,
18         beta,
19         C + p*loc, ldc);
20 }
```

Listing 2: Nested batching.

```
1 for (int q = 0; q < Q; ++q)
2 sb_gemm(OP_A, OP_B, C);
3 M, N, K,
4 1, 
5 A, lda1, lda2>, B+q*ldb2>, 0,
6 0,
7 C+q*ldc3>, ldc2>,
8 P);
9 }
```

IV. RESULTS AND DISCUSSION

In this section, we benchmark varying evaluation strategies in order to define heuristics for computing general tensor contractions without copy or transposition. Additionally,
we demonstrate the feasibility of the extended transpose parameter for exceptional case evaluations.

All performance measurements are performed on a heterogeneous CPU-GPU system with a dual-socket Intel Xeon E5-2630 v3 2.4GHz processor and an NVIDIA K40c GPU. Each CPU socket has 8 cores and 16 threads with an 8 × 256KB L2 cache and a 20MB L3 cache. The K40c has 2880 streaming cores distributed across 15 multiprocessors operating at 0.75GHz and a 1.5MB L2 cache.

All data used are randomized dense matrices. To eliminate noise from parallel competition of multi-sockets, all CPU results are generated from serial runs (one core, one thread).

A. Conventional Evaluation

We further motivate the use of STRIDEBATCHEDGEMM evaluations by plotting the speedup of the conventional approach – transpositions until a single GEMM can be called – over a single STRIDEBATCHEDGEMM call in evaluation of Case 1.3 from Table II for tensors of size \( n \times n \times n \). Figure 3 shows that STRIDEBATCHEDGEMM is significantly faster than performing even a single transposition followed by a flattened GEMM, especially for small matrices. Here, a single transposition means \( n \) calls to mkl_somatcopy on CPU or cblas_sgemm on GPU in order to fully exchange two modes. The dark lines include only a single transposition and the lighter lines include 2, 3, and 6 transpositions.

On CPU, the STRIDEBATCHEDGEMM evaluation outperforms the conventional approach for all \( n < 512 \). On GPU, the benefit from performing a single flattened GEMM eventually outweighs the cost of performing the transposition and for \( n \gtrsim 200 \) the conventional approach achieves a speedup over the STRIDEBATCHEDGEMM. This speaks to the highly optimized GEMM in CuBLAS and that, perhaps, additional optimization gains from CuBLAS’s BATCHEDGEMM may be available.

![Figure 3: Performance ratio between the conventional approach with \( \kappa \) mode transpositions over a BATCHEDGEMM in \([p]\) for Case 1.3. For color from deep to light, \( \kappa = 1, 2, 3, 6 \). Performance on CPU using MKL’s mkl_somatcopy, cblas_sgemm, and cblas_sgemm_batch. Performance on GPU using CuBLAS’s cblasSgemm, cblasSgemm, and our modified cblasSgemmBatched.](image)

B. Extended BLAS Evaluation

In this section, we compare evaluation strategies given the extended BLAS kernels. On GPU, the STRIDEBATCHEDGEMM interface is provided by modifying cblasSgemmBatched from CuBLAS 7.5. On CPU, the STRIDEBATCHEDGEMM interface is implemented in serial with looped calls to cblas_sgemm from MKL 11.2. Both implementations thereby avoid additional allocation and/or precomputation at the call site. The serial execution on CPU emphasizes the cache effects discussed in the following sections.

1) Flattening: Cases 1.1, 1.5, and 6.1 can be evaluated without explicit transpositions with either a single flattened GEMM or a single BATCHEDGEMM. We expect the flattened GEMM evaluation to outperform the BATCHEDGEMM evaluation due to the optimization level of existing GEMMs over that of the recently emerging BATCHEDGEMM functions.

In Figure 4, we plot the speedup achieved by using a flattened GEMM evaluation over a STRIDEBATCHEDGEMM evaluation. In Figure 4, the speedup is greater than one when FlattenedGEMM is faster than the STRIDEBATCHEDGEMM. Clearly, most of the time, flattened GEMM is faster. Furthermore, we note the CuBLAS implementation of STRIDEBATCHEDGEMM is a great candidate for optimization as it appears to be significantly underperforming with respect to GEMM.

We also note the dependence of the performance on the shape of the flattened GEMM and the mode of the STRIDEBATCHEDGEMM. On CPU, we find that the major determining factor in performance is the batching mode of the output. That is, the STRIDEBATCHEDGEMM evaluation performs best when batched in the third mode of \( C \) – in Case 1.5 \([p]\) and 1.1 \([p]\). On GPU, the output batching mode makes no difference. It is unclear why the batched evaluation performs so well on Case 1.5 \([p]\).

2) Batching: In this section, we attempt to quantify the performance gain by batching in the last mode versus an earlier mode and whether the input tensor or the output tensor should be prioritized for this optimization.

Case 1.1 and 2.1 can both be batched in the second \([n]\) or third \([p]\) mode. In Figure 5, we plot the speedup in...

![Figure 4: Performance ratio for a BATCHEDGEMM over a flattened GEMM in evaluation of Cases 1.1, 1.5, and 6.1. (Left) CPU. (Right) GPU.](image)
performing the BatchedGEMM in \([p]\) over performing it in \([n]\). When the size of the tensor is small, \(n \lesssim 256\), batching in the third mode is advantageous and can result in up to 1.25x speedup on CPU. When \(n \gtrsim 256\), it is approximately 1.1x faster to batch in the second mode rather than the third. We expect this is an effect of the 256KB L1 cache, which would house the contiguous \(B_{kn}\) submatrix for each \(p\) when \(n \lesssim 256\). Beyond that size both batching strategies will have forced cache misses within each GEMM, but by batching in the middle mode more data is shared between individual GEMMS.

On GPU, we see no discernible preference in the choice of batching mode. The GPU has a much less sophisticated memory system with no prefetcher and the performance difference is primarily determined by the number of global memory transactions issued. When \(n \geq 32\), the coalescing width is reached so nearly the same number of transactions will be issued in each case—with small differences caused by alignment. We confirmed this by profiling the number of global memory reads and writes issued by each kernel and verifying that they correlate with the small differences in performance observed.

Additionally, we consider the mixed-mode batching evaluations to determine if the input or output array is the primary determinant of batching performance. In Figure 6, we plot the speedup in performing StridedBatchedGEMM in the last mode of the output but the middle mode of the input, \([p]\), over performing it in the middle mode of the output and the last mode of the input, \([n]\), for Cases 1.2 and 2.2. The results are very similar to those of Figure 5 indicating that batching mode of the output tensor \(C\) is more important than the batching mode of the input tensor \(B\) on CPU. This is consistent with reference implementations of GEMM which accumulate results directly into the output matrix.

3) Exceptional Cases: In this section, we demonstrate the feasibility of evaluation strategies for the exceptional cases.

The Polyhedral Parallel Code Generator (PPCG) [28] is a source-to-source compiler capable of generating CUDA kernels from nested control loops in C. We use PPCG to generate a CUDA kernel for exceptional Case 6.4 and compare its performance against other evaluation strategies.

First, Case 6.4 has four nested loops and PPCG accepts a tiling parameter for each. We search the parameter space \((m, n, p, k) \in [1, 2, 4, 8, 16, 32, 64, 128]^4\) for the most efficient variant in Figure 7. The kernels were generated with \(\alpha = 1\) and \(\beta = 0\) statically known as generated versions with dynamic \(\alpha, \beta\) had significant branching and divergent overhead, whereas we are primarily interested in the access patterns and tiling.

The tiling parameters that result in the highest performance are \((16, 4, 32, 4)\). Via inspection, we verify that the generated kernel is performing a 2D shared memory tiling for \(A\), a “3D” shared memory tiling for \(B\), and accumulating the \(C\) results in registers.

Using the \((16, 4, 32, 4)\) kernel, we benchmark against two possible evaluation strategies: (1) A BatchedGEMM which requires no explicit transposition, and (2) A mode transposition in \(k\) and \(m\) followed by a BatchedGEMM in \([n]\). In Figure 8, we show the execution time for each with the explicit transposition/GEMM stacked to show their relative proportion in the two-step evaluation. The PPCG kernel
outperforms the explicit transposition/GEMM evaluation for small matrices and remains within a factor of 2-3x as \( n \) grows. We expect an expert implementation of the extended transpose parameter kernel would be able to close this gap and remain competitive with BATCHEDGEMM for all \( n \).

**C. Machine Learning Application**

In this section, we present the benchmarking results for the application that we discussed in Section II-C. For simulations on the CPU, we compare the performance of the Tucker decomposition using TensorToolbox, BTAS, CYCLOPS and our STRIDEDBATCHEDGEMM. For simulations on the GPU, we don’t have available GPU library to compare with, so we just evaluate our GPU implementation against STRIDEDBATCHEDGEMM. We fix the number of iterations as \( T = 200 \), set the core tensor size as \( i = j = k = 10 \), and set the dimensions as \( m = n = p \). From Figure 9, using our CPU STRIDEDBATCHEDGEMM, we obtain more than 10 times speedup compared to CYCLOPS/TensorToolbox and almost four orders of magnitude compared to BTAS. Also, as expected, our GPU STRIDEDBATCHEDGEMM confers further speedup.

**D. Evaluation Priorities**

Rather than attempt to model the algorithm and machine as in [22], [19], we simply provide evaluation guidelines based on the data provided. These are a number of heuristics that may be important in constructing the most efficient evaluation strategy.

1) Flatten modes whenever possible. A single large GEMM is more efficient.

2) In the interest of performing the highest intensity computation within a BATCHEDGEMM, we recommend performing the largest GEMMs possible within a BATCHEDGEMM and batching in the mode with largest dimension.

3) Preferring to batch in the last mode versus earlier modes can depend on the input parameters and machine.

We summarize these evaluation guidelines with pseudocode for performing a single-index tensor contraction without copy or transposition in Algorithm 2.

**V. CONCLUSIONS AND FUTURE WORK**

Our experience reveals that the emergence of BATCHEDGEMM provides significant computational advantages for multi-linear algebraic computations. The primitive allows us to push a larger high intensity computations to vendor-provided implementations. Leading implementations already provide BATCHEDGEMM on highly parallel machines. To simplify their use and provide additional optimization opportunities, we propose STRIDEDBATCHEDGEMM and demonstrate its use for generalized tensor contractions. Calls to STRIDEDBATCHEDGEMM have significant opportunity to perform at or near the performance of GEMM and, by avoiding explicit transpositions or permutations of the data, accelerate these computations significantly.

Our improvement is most significant on small and moderate sized tensors. This is very important because in many applications, e.g. deep learning for training a recursive tensor network, we require evaluating a large number of tensor contractions of small sizes.

Although we focused on single-node performance, these evaluations may be used as building blocks for distributed memory implementations, which we intend to pursue as part of our future work. Further study into the optimized implementations, architecture-dependent implementations, and performance of the exceptional case kernels is warranted. More complicated contractions, such as multi-index contractions or sparse tensor algebra, also pose challenging problems.

**ACKNOWLEDGMENT**

The authors would like to thank Aparna Chandramowlishwaran for providing the computation resources and suggestions. Animashree Anandkumar is supported in part by Microsoft Faculty Fellowship, NSF Career Award CCF-1254106, ONR Award N00014-14-1-0665, ARO YIP Award W911NF-13-1-0084, and AFOSR YIP FA9550-15-1-0221. Yang Shi is supported by NSF Career Award CCF-1254106 and ONR Award N00014-15-1-2737, Niranjan is supported by NSF BigData Award IIS-1251267 and ONR Award N00014-15-1-2737.
Algorithm 2 Single-Mode Tensor Contraction

1: In: Tensor $A_A$, $A = [a_1, \ldots, a_M]$.  
2: In: Tensor $B_B$, $B = [b_1, \ldots, b_N]$, $A \cap B = \{k\}$.  
3: In, Out: Tensor $C_C$, $C = [c_1, \ldots, c_{N+M-2}]$. WLOG, $c_1 \in A$.  
4: Common substrings in $A$, $B$ and/or $C$ for flattening candidates.  
5: Relabel flattened modes  
6: Compute $\mathcal{P} = \{c_j \mid i \neq 1, c_i \neq a_1, c_i \neq b_1\}$  
7: if $|\mathcal{P} \setminus \{c\}| = 1$, then  
8: $[\text{Case } C_{c_1\ldots c}\ldots = A_{k} \times A_{c_1\ldots c}\ldots B_k\ldots]$  
9: Let $c^* \in \mathcal{P} \setminus A$ be index with max dimension  
10: Let $c^* \in \mathcal{P} \setminus \{c\}$ be index with max dimension  
11: Nested in all $c_j \in \mathcal{P} \setminus \{c^*, c_1\}$, BATCHEDGEMM in $c_1, c^*, k, [c^*]$  
12: else if $|\mathcal{P} \setminus \{c\}| = 2$, then  
13: $[\text{Case } C_{c_1\ldots c_2\ldots} = A_{k} \times c_1 \times c_2 \times B_{k} \times \ldots]$  
14: Let $c^* \in \mathcal{P}$ be index with max dimension  
15: Nested in all $c_j \in \mathcal{P} \setminus \{c^*\}$, BATCHEDGEMM in $c_1, c_2, k, [c^*]$  
16: else if $|\mathcal{P} \setminus \{c\}| = 2$, then  
17: $[\text{Case } C_{c_1\ldots c_2\ldots} = A_{k} \times c_1 \times c_2 \times B_{k} \ldots]$  
18: Let $c^* \in \mathcal{P} \setminus A$ be index with max dimension  
19: Nested in all $c_j \in \mathcal{P} \setminus \{c^*\}$, EX. BATCHEDGEMM in $c_1, c^*, k, [c^*]$  
20: else if $|\mathcal{P} \setminus \{c\}| = 3$, then  
21: $[\text{Case } C_{c_1\ldots c_2\ldots c_3\ldots} = A_{k} \times c_1 \times c_3 \times B_{k} \ldots]$  
22: Nested in all $c_j \in \mathcal{P}$, EX. BATCHEDGEMM in $c_1, c_2, k, [c^*]$  
23: end if

References


