An Input-Adaptive and In-Place Approach to Dense Tensor-Times-Matrix Multiply

Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, Richard Vuduc
Georgia Institute of Technology
North Ave NW
Atlanta, GA 30332
{jiajiali, cbattaglino3, perros}@gatech.edu {jsun, richie}@cc.gatech.edu,

ABSTRACT

This paper describes a novel framework, called InTensLi ("intensely"), for producing fast single-node implementations of dense tensor-times-matrix multiply (Ttm) of arbitrary dimension. Whereas conventional implementations of Ttm rely on explicitly converting the input tensor operand into a matrix—in order to be able to use any available and fast general matrix-matrix multiply (Gemm) implementation—our framework's strategy is to carry out the Ttm *in-place*, avoiding this copy. As the resulting implementations expose tuning parameters, this paper also describes a heuristic empirical model for selecting an optimal configuration based on the Ttm's inputs. When compared to widely used single-node Ttm implementations that are available in the Tensor Toolbox and Cyclops Tensor Framework (Ctf), InTensLi's in-place and input-adaptive Ttm implementations achieve 4× and 13× speedups, showing Gemm-like performance on a variety of input sizes.

Categories and Subject Descriptors

G.1.0 [**Mathematics of Computing**]: Numerical Analysis—*Numerical algorithms*; C.1.2 [**Computer Systems Organization**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors*

Keywords

Multilinear algebra, tensor operation, code generation, offline autotuning

1. INTRODUCTION

We consider the problem of how to improve the single-node performance of a dense *mode-n product* [21], or more simply, a dense *tensor-times-matrix multiply (*Ttm*)*. One may regard a tensor as the multidimensional generalization of a matrix, as depicted in figure 1 and formalized in § 2; a dense Ttm multiplies a dense tensor by a dense matrix.

SC '15, November 15-20, 2015, Austin, TX, USA

 ISBN 978-1-4503-3723-6/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807671

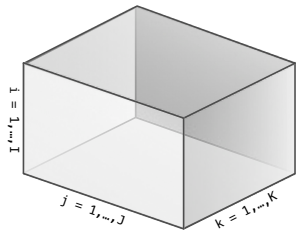


Figure 1: A third-order (or 3-way) tensor may be thought of as a 3-dimensional array, or as a 3-D analogue of a 2-D matrix.

Tensor-based computations, or *multilinear algebraic* computations, underlie a variety of current and emerging applications in chemometrics, quantum chemistry and physics, signal and image processing, neuroscience, and data analytics, to name a few [2–4, 10, 15, 15, 19, 20, 23, 24, 26–29, 31, 34–37, 41, 44]. For such computations, Ttm is both a fundamental building block and usually the dominant bottleneck.

The conventional way to implement Ttm relies on *matricization* of the tensor, which essentially transforms the tensor into an equivalent matrix [21]. Ttm then simply becomes a general matrix-matrix multiply (Gemm). This approach is sensible, for two reasons. First, Gemm is a clean and portable abstraction for which many fast implementations already exist [11, 17, 43, 45]. Secondly, the basic theoretical computation and communication complexity of Ttm matches Gemm [39]. Thus, Ttm has high arithmetic intensity at sufficiently large sizes and so should scale well, just like Gemm.

Unfortunately, we observe that state-of-the-art Gemm-based implementations of Ttm, such as the Tensor Toolbox or Cyclops Tensor Framework (Ctf) [40], can perform well *below* what one expects from Gemm alone (§ 3). The problem is matricization.

Mathematically, matricization is merely a conceptual (or logical) restructuring of the tensor. However, this step is almost always implemented as an explicit copy. Copying ensures that the matricized tensor has a non-unit stride in only one dimension, as required by the Basic Linear Algebra Subprograms (Blas) [1]. By contrast, doing the Ttm in-place *without* such copies would require a different interface for Gemm, one which could support non-unit strides in *both* the row and column dimensions [43]; accordingly, performance may be expected to decrease due to decreases in cache line utilization, prefetch bandwidth, and the degree of

SIMDization. (Copying can also roughly double the storage; see § 3.) Nevertheless, the conventional wisdom is that this copy need not incur a *performance* overhead. By contrast, we find that explicit matricization can in practice *dominate* the overall running time, accounting for 70% of the total running time or more, even at large sizes[1] (figure 4 in § 3).

These observations raise a very natural question: can one achieve GEMM-like performance for TTM using only an *in-place* approach?

Contributions.

We answer the preceding question affirmatively. Our main contribution is a novel framework for automatically generating high-performance in-place TTM implementations (§ 4). This framework generates TTM implementations that use coarse-grained parallelism via OpenMP and any underlying high-performance GEMM. It considers several strategies for decomposing the specific type of TTM that the user requires in order to maximize the use of the fast GEMM. We show that our framework can produce TTM implementations that achieve a high fraction of what a hypothetical pure GEMM can achieve (§ 5).

Beyond a code generation strategy, the second contribution of our work is a method, based on empirical model-based tuning, to select a good implementation of the TTM. Our method considers the characteristics of the input to the TTM, such as the size of the tensor in each dimension. Put differently, the code generation framework produces several parameterized implementations; and we use a heuristic model to select the parameters, which need to be tuned for a given input. In this way, the framework's results are *input-adaptive* [25]. Thus, even if a compiler-based loop transformation system can, from a naïve TTM code as input, produce the codes that our framework generates, the input-adaptive aspect of our approach can be considered a distinct contribution.

We have implemented our approach as a system called INTENSLI, which is pronounced as "intensely" and is intended to evoke an *In-place and input-adaptive Tensor Library*.

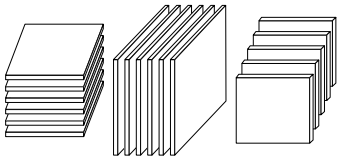
Taken together, we show that our INTENSLI-generated TTM codes can outperform the TTM implementations available in two widely used tools, the TENSOR TOOLBOX [21] and CTF [40], by about 4 times and 13 times, respectively. Our framework can be directly applied to tensor decompositions, such as the well-known Tucker decomposition, whose computation heavily relies on efficient tensor-times-matrix multiply [21].

2. BACKGROUND

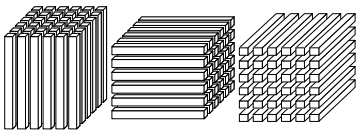
This section provides some essential formal background on tensors. To help keep this paper relatively self-contained, several of the examples and definitions are taken from the excellent overview by Kolda and Bader [21].

A tensor can be interpreted as a multi-way array. This fact is illustrated graphically in figure 1, which shows a 3rd-order tensor. We represent a tensor by a bold underlined capital letter, e.g., $\underline{\mathbf{X}} \in \mathbb{R}^{I \times J \times K}$. The *order* of a tensor is the number of its dimensions or modes, which in the example is 3. Matrices and vectors are special cases of tensors. Matrices are 2nd-order tensors, and we denote them by boldface capital letters, e.g., $\mathbf{A} \in \mathbb{R}^{I \times J}$. Vectors are 1st-order tensors, and we denote them by boldface lowercase letters, e.g., $\mathbf{x}$. The elements of a tensor are scalars, and we denote them by lowercase letters, such as x_{ijk} for the (i, j, k) element of a 3rd-order tensor $\underline{\mathbf{X}}$.

Given a tensor, we can also ask for a variety of *sub-tensors*. One form of sub-tensor is a *slice*, which is a 2-dimensional cross-section of a tensor. Figure 2(a) illustrates the concept of slices, showing different slices of a 3rd-order tensor. Another form of sub-tensor is a *fiber*, illustrated in figure 2(b). A fiber is a vector extracted from the tensor, and is defined by fixing every index but one [21]. Figure 2(b) gives three fibers of the tensor, denoted by $\mathbf{x}_{:jk}, \mathbf{x}_{i:k}, \mathbf{x}_{ij:}$. Since slices are all matrices, they can for the 3rd-order example of figure 2(a) be represented by $\mathbf{X}_{i::}$, $\mathbf{X}_{:j:}$, and $\mathbf{X}_{::k}$. In this paper, we consider double-precision real-valued elements for all tensors, matrices, and scalars.



(a) Horizontal, lateral, and frontal slices of a third-order tensor



(b) Column (mode-1), row (mode- 2), and tube (mode-3) fibers of a third order tensor.

Figure 2: Some sub-tensor views of the third-order tensor $\underline{\mathbf{X}} \in \mathbb{R}^{I \times J \times K}$ shown in figure 1.

Examples of tensor operations include the mode-n product introduced in § 1, tensor contraction, and Kronecker product [21]. Since the mode-n product is among the most widely used of these primitives, we focus on it in this paper and refer to it alternately as the tensor-times-matrix multiply (TTM), following the terminology of the TENSOR TOOLBOX [6]. Comprehensive surveys of other tensor operations appear elsewhere [10, 21].

The TTM between a tensor $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and a matrix $\mathbf{U} \in \mathbb{R}^{J \times I_n}$ is another tensor $\underline{\mathbf{Y}} = \underline{\mathbf{X}} \times_n \mathbf{U}$, where $\underline{\mathbf{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N}$. One way to precisely define TTM is to express it as the element-wise computation,

$$\begin{aligned} y_{i_1 \ldots i_{n-1} j i_{n+1} \ldots i_N} &= (\underline{\mathbf{X}} \times_n \mathbf{U})_{i_1 \ldots i_{n-1} j i_{n+1} \ldots i_N} \\ &= \sum_{i_n=1}^{I_n} x_{i_1 i_2 \ldots i_N} u_{j i_n}. \end{aligned} \tag{1}$$

Note that where the input tensor $\underline{\mathbf{X}}$ is of length I_n in its n^{th} mode, the result $\underline{\mathbf{Y}}$ is of length J in its n^{th} mode. Typically, J will be much less than I_n, which we show has important consequences for performance in § 3.

[1] The argument would be that the copy is, asymptotically, a negligible low-order term in the overall running time. However, that argument does not hold in many common cases, such as the case of the output tensor being is much smaller than the input tensor. This case arises in machine learning and other data analysis applications, and the result is that the cost of a data copy is no longer negligible.

The traditional way to execute a TTM is through *matricization*, also called *unfolding* or *flattening* in the literature. That is, TTM is equivalent to a matrix-matrix multiplication in the form,

$$\underline{\mathbf{Y}} = \underline{\mathbf{X}} \times_n \mathbf{U} \Leftrightarrow \mathbf{Y}_{(n)} = \mathbf{U}\mathbf{X}_{(n)}, \tag{2}$$

where $\mathbf{Y}_{(n)}$ and $\mathbf{X}_{(n)}$ denote the matricized forms of $\underline{\mathbf{Y}}$ and $\underline{\mathbf{X}}$, respectively. For a mode-n product, matricization physically reorders the elements of an order-N tensor into a matrix, by exchanging mode-n with the leading dimension that is dictated by the matrix data layout. (For instance, if the data is stored in row-major layout, then the last dimension (mode-N) is the leading dimension.) As an example, a $3 \times 4 \times 2$ tensor with elements from 1 to 24 can be arranged as a 3×8 matrix, or a 4×6 matrix, or a 2×12 matrix. In the context of tensors (rather than performance optimization), the *vectorization* operation, $\mathbf{y} = \text{vec}(\underline{\mathbf{X}})$, converts a tensor into an equivalent vector. The reverse process of creating a tensor from a matrix or vector is called *tensorization*.

$$\mathbf{X}_{(1)} = \begin{bmatrix} 1 & 4 & 7 & 10 & 13 & 16 & 19 & 22 \\ 2 & 5 & 8 & 11 & 14 & 17 & 20 & 23 \\ 3 & 6 & 9 & 12 & 15 & 18 & 21 & 24 \end{bmatrix},$$

$$\mathbf{X}_{(2)} = \begin{bmatrix} 1 & 2 & 3 & 13 & 14 & 15 \\ 4 & 5 & 6 & 16 & 17 & 18 \\ 7 & 8 & 9 & 19 & 20 & 21 \\ 10 & 11 & 12 & 22 & 23 & 24 \end{bmatrix}, \tag{3}$$

$$\mathbf{X}_{(3)} = \begin{bmatrix} 1 & 2 & 3 & \dots & 10 & 11 & 12 \\ 13 & 14 & 15 & \dots & 22 & 23 & 24 \end{bmatrix}.$$

Currently, most tensor toolkits, including the widely used TENSOR TOOLBOX and CTF libraries [6, 40], implement TTM using a three-step method: (i) matricizing the tensor by *physically* reorganizing it (copying it) into a matrix; (ii) carrying out the matrix-matrix product, using a fast GEMM implementation; and (iii) tensorizing the result, again by physical reorganization. This procedure appears in algorithm 1 and figure 3. We analyze this procedure in more detail in § 3.

Input: A dense tensor $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a matrix $\mathbf{U} \in \mathbb{R}^{J \times I_n}$, and an integer n;
Output: A dense tensor $\underline{\mathbf{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N}$;
// **Matricize**: Transform from $\underline{\mathbf{X}}$ to $\mathbf{X}_{(n)}$.
1: Set sz = size($\underline{\mathbf{X}}$), order = $[n,\ 1:n-1,\ n+1:N]$;
2: $\widetilde{\underline{\mathbf{X}}} = \text{permute}(\underline{\mathbf{X}}, \text{order})$;
3: $\mathbf{X}_{\text{mat}} = \text{reshape}(\widetilde{\underline{\mathbf{X}}})$, $\mathbf{X}_{\text{mat}} \in \mathbb{R}^{I_n \times \Pi_{-n}^{i=1,\dots,N} I_i}$;
// **Multiply**: $\mathbf{Y}_{(n)} = \mathbf{U}\mathbf{X}_{(n)}$.
4: $\mathbf{Y}_{\text{mat}} = \mathbf{U} * \mathbf{X}_{\text{mat}}$;
// **Tensorize**: Transform from $\mathbf{X}_{(n)}$ to $\underline{\mathbf{X}}$.
5: Set out_sz = $[J, \text{sz}(1:n-1), \text{sz}(n+1:N)]$;
6: $\widetilde{\underline{\mathbf{Y}}} = \text{tensor}(\mathbf{Y}_{\text{mat}}, \text{out\_sz})$;
7: $\underline{\mathbf{Y}} = \text{inversePermute}(\widetilde{\underline{\mathbf{Y}}}, \text{order})$;
8: **return** $\underline{\mathbf{Y}}$;

Algorithm 1: The baseline mode-n product algorithm (TTM) in Tensor Toolbox [6] and CTF [40].

One particular motivation for improving TTM performance is its central role in the tensor decomposition, widely used to create a low-rank representation of a tensor [13], as a higher-dimensional generalization of the singular value decomposition (SVD). The higher-order orthogonal iteration (HOOI) variant, or TUCKER-HOOI algorithm, computes for each mode n

$$\underline{\mathbf{Y}} = \underline{\mathbf{X}} \times_1 \mathbf{A}^{(1)T} \cdots \times_{n-1} \mathbf{A}^{(n-1)T} \times_{n+1} \mathbf{A}^{(n+1)T} \cdots \times_N \mathbf{A}^{(N)T}$$

until convergence occurs. Thus, we have $n(n-1)$ mode-n products over the course of a single iteration.

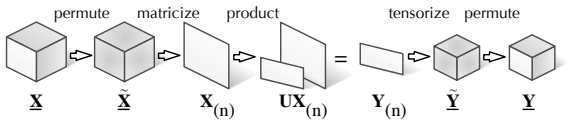
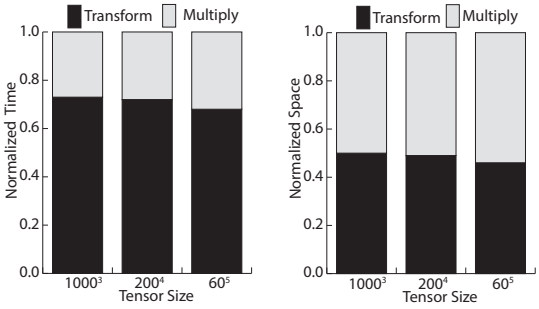


Figure 3: Structure of the baseline TTM computation.

3. MOTIVATING OBSERVATIONS

To motivate the techniques proposed in § 4, we make a few observations about the traditional implementations of TTM (algorithm 1). These observations reveal several interesting problems and immediately suggest possible ways to improve TTM performance.



(a) Time Profiling (b) Space Profiling

Figure 4: Profiling of algorithm 1 for mode-2 product on 3rd, 4th, and 5th-order tensors, where the output tensors are low-rank representations of corresponding input tensors.

Observation 1: *In a traditional* TTM *implementation, copy overheads dominate when the output tensor is much smaller than the input tensor.* Consider algorithm 1, which has two transformation steps, one on lines 2–3 and the other on lines 6–7. They can carry a large overhead when implemented as physical reorganizations, i.e., explicit copies.

For instance, consider the profiling results for a mode-2 product on 3rd-order, 4th-order, and 5th-order dense tensors, as shown in figure 4. The x-axis represents tensor sizes and the y-axis shows the fraction of time or space of the transformation step (lower dark bars) compared to the matrix multiply (upper light-grey bars). These profiles were gathered for the TENSOR TOOLBOX implementation.[2] Transformation takes about 70% of the total running time and accounts for 50% of the total storage.

[2]In this paper, TENSOR TOOLBOX calls Intel Math Kernel Library as the high-performance BLAS implementation of GEMM.

To see fundamentally why copying hurts performance, compare the arithmetic intensity of TTM with copy overheads against GEMM. Consider a system with a two-level memory hierarchy, that is, with a large slow memory connected to a fast memory (e.g., last-level cache) of size Z words. Recall that the *arithmetic intensity* of a computation running on such a system is the ratio of its floating-point operations, or *flops* Q, to the number of words, W, moved between slow and fast memory [16]; the higher a computation's intensity, the closer to peak it will run. For both GEMM and tensor contractions, $W \geq \frac{Q}{8\sqrt{Z}} - Z$ [7]. Thus, an upper-bound on arithmetic intensity, A, is

$$A \equiv \frac{Q}{W} \quad \leq \quad \frac{Q}{\frac{Q}{8\sqrt{Z}} - Z} \approx 8\sqrt{Z}, \tag{4}$$

where the latter bound holds if $\frac{Q}{8\sqrt{Z}} \gg Z$, or $Q \gg 8Z^{3/2}$. For a non-Strassen GEMM on $n \times n$ matrices, $Q = 2n^3$, so that equation (4) holds when $n^3 \gg 4Z^{3/2}$. Assuming a cache of size 8 MiB, which is 2^{20} double-precision words, then $4Z^{3/2} = 2^{32}$, which means the matrix dimension n should satisfy $n \gtrsim 1600$.

Now consider a TTM implementation that copies explicitly. Suppose this TTM involves an order-d tensor of size m in each dimension, so that the tensor's size is m^d and the TTM's flops are $\hat{Q} = 2m^{d+1}$. The two transformations steps together require moving a total of $2m^d$ words. Further suppose that this TTM does the *same* number of flops as the preceding GEMM; then, $\hat{Q} = Q$ or $m = n^{\frac{3}{d+1}}$. The intensity $\hat{A}$ of this TTM will be, again assuming $\hat{Q} = Q \gg 8Z^{3/2}$,

$$\hat{A} \quad \lesssim \quad \frac{\hat{Q}}{\frac{\hat{Q}}{8\sqrt{Z}} + \frac{\hat{Q}}{m}} = \frac{8\sqrt{Z}}{1 + \frac{8\sqrt{Z}}{m}} \approx \frac{A}{1 + \frac{A}{m}}. \tag{5}$$

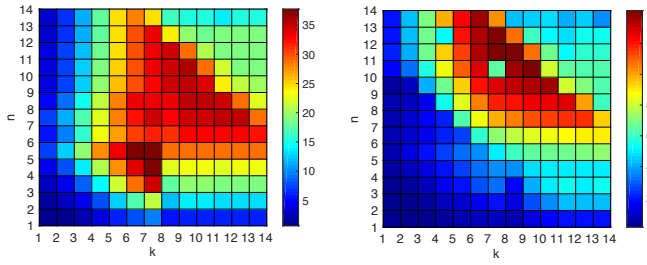
Thus, copying might reduce the copy-free intensity, A, by a factor of $1 + A/m$. How large can this penalty be? If $n \gtrsim 1600$ and $d = 3$, then $m \approx 254$ and $1 + A/m \approx 33$. This penalty increases as the order d increases. Thus, one should avoid explicit copying.

Observation 2: *The* GEMM *sizes that arise in* TTM *may be far below peak.* It is well-known that while GEMM benchmarks emphasize its performance on large and square problem sizes, the problem sizes that arise in practical applications are often rectangular and, therefore, may have a very different performance profile. The best example is LU decomposition, in which one repeatedly multiples a "tall-skinny" matrix by a "short-fat" matrix.[3] Similarly, one expects rectangular sizes to arise in TTM as well, due to the nature of the decomposition algorithms that invoke it. That is, the GEMM on line 4 of algorithm 1 will multiply operands of varying and rectangular sizes. A particularly common case is one in which $J \ll I_i$, which can reduce intensity relative to the case of $J \approx I_i$.

Based on these expectations, we measured the performance of a highly-tuned GEMM implementation from Intel's Math Kernel Library. Figure 5 shows the results. The operation was $\mathbf{C} = \mathbf{B}\mathbf{A}^T$, where $\mathbf{A} \in \mathbb{R}^{n \times k}$, $\mathbf{B} \in \mathbb{R}^{m \times k}$. The value of m is fixed to be $m = 16$, which corresponds to a "small" value of J. Each square is the performance at a particular value of k (x-axis, shown as $\log_2 k$) and n (y-axis, shown as $\log_2 n$), color-coded by its performance in GFLOP/s. Figure 5(a) shows the case of a single thread, and (b) for 4 threads. (The system is a single-socket quad-core platform, described in § 5.)

[3] That is, block outer products.

According to figure 5, performance can vary by roughly a factor of 6, depending on the operand sizes. When k or n becomes very large, performance can actually decrease. The maximum performance in figure 5 is about 38 GFLOP/s for 1 thread, and 140 GFLOP/s for 4 threads. This particular GEMM can actually achieve up to 50 GFLOP/s and 185 GFLOP/s for 1 and 4 threads, respectively, at large square sizes.



(a) m=16, using 1 thread (b) m=16, using 4 threads

Figure 5: Performance of dense, double-precision general matrix multiply from the Intel Math Kernel Library for $\mathbf{C} = \mathbf{B}\mathbf{A}^T$, where $\mathbf{A}$ is $n \times k$, $\mathbf{B}$ is $m \times k$. The value of m is fixed at 16, while k (x-axis) and n (y-axis) vary. Note that the x- and y-axis labels show $\log_2 k$ and $\log_2 n$, respectively. Each square is color-coded by performance in GFLOP/s.

Observation 3: *Analogous to* BLAS *operations, different ways of organizing* TTM *have inherently different levels of locality.* There are several ways to organize a TTM. Some of these formulations lead to scalar operations, which have relatively poor locality compared to Level-2 (matrix-vector) or to Level-3 (matrix-matrix) type operations. As with linear algebra operations, one should prefer Level-3 organizations over Level-2 or Level-1.

We summarize these different formulations in table 1, for the specific example of computing a mode-1 TTM on a 3rd-order tensor. There are two major categories. The first is based on *full reorganization* (algorithm 1), which transforms the entire input tensor into another form before carrying out the product. This category includes the matricization approach. The second category is based on *sub-tensor extraction*. The formulations in this category iterate over sub-tensors of different shapes, such as scalars, fibers, or slices (§ 2). By analogy to the BLAS, each of the formulations corresponds to the BLAS level shown in the table (column 3). The full reorganizations also imply at least a logical transformation of the input tensor, whereas sub-tensor extraction methods do not (column 4). Further explanation of different representations will be given in § 4.

4. IN-PLACE AND INPUT-ADAPTIVE TTM

This section describes the INTENSLI framework, which automatically generates an in-place, input-adpative TTM (InTTM). By in-place, we mean avoiding the need for physical reorganization of the tensor or its sub-tensors when computing the TTM; by input-adaptive, we mean choosing a near-optimal strategy for a given set of inputs, which for a mode-n product include the value of n and the size of the input tensor (both order and length of each dimension).

Table 1: A 3rd-order tensor's different representation forms of mode-1 Ttm. The input tensor is $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, the input matrix is $\mathbf{U} \in \mathbb{R}^{J \times I_1}$, and the output tensor is $\underline{\mathbf{Y}} \in \mathbb{R}^{J \times I_2 \times I_3}$.

Mode-1 Product Representation Forms		BLAS Level	Transformation
Full reorganization	*Tensor representation* $\underline{\mathbf{Y}} = \underline{\mathbf{X}} \times_1 \mathbf{U}$	—	—
	Matrix representation $\mathbf{Y}_{(1)} = \mathbf{U}\mathbf{X}_{(1)}$	L3	Yes
Sub-tensor extraction	*Scalar representation* $y_{ji_2i_3} = \sum_{i_1=1}^{I_1} x_{i_1i_2i_3} u_{ji_1}$, *Loops* : $i_{2(3)} = 1, \cdots, I_{2(3)}, j = 1, \cdots, J$	Slow	No
	Fiber representation $\mathbf{y}(j, :, i_3) = \mathbf{X}(:, :, i_3)\mathbf{u}(j, :)$, *Loops* : $i_3 = 1, \cdots, I_3, j = 1, \cdots, J$	L2	No
	Slice representation $\mathbf{Y}(:, :, i_3) = \mathbf{U}\mathbf{X}(:, :, i_3)$, *Loops* : $i_3 = 1, \cdots, I_3$	L3	No

4.1 A 3rd-order tensor example

To build some intuition for our basic scheme, it is helpful to introduce some basic terms on a concrete example.

Consider a three-dimensional tensor $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ and a matrix $\mathbf{U} \in \mathbb{R}^{J \times I_n}$, with which we wish to compute the mode-n product (or TTM), $\underline{\mathbf{Y}} = \underline{\mathbf{X}} \times_n \mathbf{U}$. From the discussion of § 3 and table 1, recall that there are several ways to evaluate a TTM. For instance, table 1 includes several forms, such as the slice representation, which fixes one mode of $\underline{\mathbf{X}}$; the fiber representation, which fixes two; and the scalar representation, which fixes all modes inside the nested loops. These operations can be performed *in-place* so long as we pre-allocate $\underline{\mathbf{Y}}$. Moreover, having chosen a form we are also free to choose which dimensions to fix and which to iterate over. Alternatively, we may compute $\underline{\mathbf{Y}}$ by slices by computing, say, $\mathbf{Y}(:, i_2, :) = \mathbf{U}\mathbf{X}(:, i_2, :)$ for all $i_2 = 1, \cdots, I_2$, instead of fixing mode-3 in table 1.

The term *stride* refers to the distance between consecutive elements along the same mode. The *leading dimension* is the dimension with unit stride, or stride equal to 1. For instance, suppose our three-dimensional tensor $\underline{\mathbf{X}}$ is stored in *row-major layout*. Then, each element x_{ijk} is stored at position $i \cdot (I_2 I_3) + j \cdot (I_3) + k$. The leading dimension is the third (i_3) dimension. We take two alternatives of the above slice representations as examples, $\mathbf{Y}(:, :, i_3) = \mathbf{U}\mathbf{X}(:, :, i_3)$ and $\mathbf{Y}(:, i_2, :) = \mathbf{U}\mathbf{X}(:, i_2, :)$. The row stride of matrix $\mathbf{X}(:, :, i_3)$ (representation in table 1) is $I_2 I_3$ and the column stride is I_3, for some i_3, while the row stride of matrix $\mathbf{X}(:, i_2, :)$ (alternative representation) is $I_2 I_3$ and the column stride is 1. When both the row stride and the column stride are non-unit, the matrix is stored in a *general stride layout*.

The traditional BLAS interface for GEMM operations *require* unit stride in one dimension, and so cannot operate on operands stored with general stride. The recent BLAS-like Library Instantiation Software (BLIS) framework can export interfaces to support generalized matrix storage [43]. However, GEMM on strided matrices is generally expected to be slower than GEMM on matrices, since strided loads might utilize cache lines or other multi-word transfers more poorly than unit stride accesses. Therefore, a reasonable implementation heuristic is to avoid non-unit stride computation. For tensor computations, that also means *we should prefer to build or reference sub-tensors starting with the leading dimension*. Thus, we prefer the first slice representation (same with table 1) over the second.

We can also reshape a tensor into a matrix by partitioning its indices into two disjoint subsets, in different ways. For example, the tensor $\underline{\mathbf{X}}(:, :, :) \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ can be reshaped into a matrix $\widetilde{\mathbf{X}}(:, :) \in \mathbb{R}^{(I_1 \times I_2) \times I_3}$, that is, a matrix with $I_1 I_2$ rows and I_3 columns. This particular reshaping is purely logical: it does not require physically reorganizing the underlying storage before passing it either to a BLAS-style GEMM that requires a unit-stride matrix or a BLIS-style GEMM that supports a general stride matrix (see § 4.2). Another way to reshape $\underline{\mathbf{X}}$ is to generate a matrix $\hat{\mathbf{X}}(:, :) \in \mathbb{R}^{I_2 \times (I_1 \times I_3)}$. As it happens, this reshape operation is impossible without physically reorganizing the data (see § 4.2). *Thus, we need to be careful when choosing a new dimension order of a tensor, to avoid physical reorganization.* These two facts are the key ideas behind our overall strategy for performing TTM in-place.

4.2 Algorithmic Strategy

We state two lemmas: the first suggests how to build matrices given an arbitrary number of modes of the input tensor, and the second establishes the correctness of computing a matrix-matrix multiplication on sub-tensors.

LEMMA 4.1. *Given an Nth-order tensor $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and a matrix $\mathbf{U} \in \mathbb{R}^{J \times I_n}$, the mode-$n$ product can be performed without physical reorganization on up to $\max\{n-1, N-n\}$ contiguous dimensions.*

PROOF. From section § 4.1, to avoid physical reorganization, the dimensions of sub-tensors should be a sub-sequence of the input tensor's. Combining contiguous dimensions is the only way to build a high performance MM kernel with sub-tensors (matrices).

We first prove that two contiguous modes (except mode-n) can be reshaped to one dimension without physical reorganization. Let m and $m+1$ be two contiguous modes. Let $\widetilde{\underline{\mathbf{X}}} \in \mathbb{R}^{I_1 \times \cdots \times I_{new} \cdots \times I_N}$, where $I_{new} = I_m I_{m+1}$, be the reshaped tensor that comes from combining these two modes into a new dimension. Because the order of the two contiguous modes is unchanged, elements of tensor $\widetilde{\underline{\mathbf{X}}}$ have exactly the same physical arrangement as tensor $\underline{\mathbf{X}}$. Thus, we can logically form tensor $\widetilde{\underline{\mathbf{X}}}$ from $\underline{\mathbf{X}}$ without data movement. However, for two non-contiguous modes, e.g. mode-m and $(m+2)$, we cannot form a new reshaped tensor without a permutation, which demands physically reorganizing $\underline{\mathbf{X}}$.

When reshaping contiguous modes, the resulting sub-tensors $\mathbf{X}_{\text{sub}} \in \mathbb{R}^{I_n \times I_{new}}$ are actually matrices. For a mode-n prod-

uct, the most contiguous modes are obtained by either the leftmost modes of mode-n, $\{1, \cdots, n-1\}$ or the rightmost ones $\{n+1, \cdots, N\}$. Thus, up to $\max\{n-1, N-n\}$ contiguous modes can be reshaped into a "long" single dimension in the formed matrices, without physical data reorganization. □

Lemma 4.2 applies lemma 4.1 to the computation of the mode-n product.

LEMMA 4.2. *Consider the mode-n product, involving an order-N tensor. It can be computed by a sequence of matrix multiplies involving sub-tensors formed either from the leftmost contiguous modes, $\{1, \ldots, m_1\}$, $1 \le m_1 \le n-1$; or the rightmost contiguous modes, $\{m_2, \ldots, N\}$, $n+1 \le m_2 \le N$.*

PROOF. Recall the scalar definition of the mode-n product, $\underline{\mathbf{Y}} = \underline{\mathbf{X}} \times_n \mathbf{U}$, which is

$$y_{i_1 \cdots i_{n-1} j \cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 \cdots i_N} u_{j i_n}.$$

Without loss of generality, consider the rightmost contiguous modes $\{m_2, \cdots, N\}$, and suppose they are combined into a single dimension. Let $I_p \equiv I_{m_2} \cdots I_N$ be the total size of this new dimension. Then, we may extract two new (logical) sub-tensors (matrices): $\mathbf{X}_{\text{sub}}$, which is taken from $\underline{\mathbf{X}}$ and has size $I_n \times I_p$, and $\mathbf{Y}_{\text{sub}}$, which is taken from $\underline{\mathbf{Y}}$ and has size $J \times I_p$. The GEMM operation, $\mathbf{Y}_{\text{sub}} = \mathbf{U}\mathbf{X}_{\text{sub}}$, is in scalar form, $(Y_{\text{sub}})_{j i_p} = \sum_{i_n=1}^{I_n} u_{j i_n} (X_{\text{sub}})_{i_n, i_p}$. The index $i_p = i_{m_2} \times I_{m_2+1} \cdots I_N + \cdots + i_N$, corresponds to the offset $(i_{m_2}, \cdots, i_N)$ in the tensors $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$; the remaining modes, $\{i_1, \cdots, i_{n-1}, i_{n+1}, \cdots, i_{m_2-1}\}$, are fixed indices during this GEMM. Thus, iterating over all possible values of $\{i_1, \cdots, i_{n-1}, i_{n+1}, \cdots, i_{m_2-1}\}$ and performing the corresponding matrix multiplies yields the same result as the mode-n product. □

Lemma 4.1 and lemma 4.2 imply an alternative representation of a mode-n product, relative to those listed in table 1. This algorithmic strategy may be summarized by the in-place tensor-times-matrix multiply (INTTM) procedure shown in algorithm 2 and figure 6.

To instantiate this algorithm for a tensor with given dimensions, we must generate a set of nested loops (line 1) and use a proper kernel for the inner matrix-matrix multiplication (lines 5 and 9).

A number of parameters arise as a result of this algorithm:

- *Loop modes, M_L.* Modes utilized in nested loops. They are not involved in the inner-most matrix multiplies. For instance, in table 1, the loop mode is i_3 in the "Slice representation."
- *Component modes, M_C.* Modes participate in matrix multiply. M_L and M_C constitute the set of all tensor modes, except mode-n. From lemma 4.1, only contiguous modes are considered as component modes. In table 1, component mode is i_2.
- *Loop parallel degree, P_L.* The number of threads utilized by nested loops, i.e., loop modes, to exploit parallelization at a coarse-grained level.
- *MM parallel degree, P_C.* The number of threads employed by the inner-most matrix multiply; fine-grained parallelism.

Input: A dense tensor $\underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{J \times I_n}$, and an integer n;
Output: A dense tensor $\underline{\mathbf{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N}$;

// Nested loops, using P_L threads
1: **parfor** $i_l = 1$ to I_l, all $i_l \in M_L$ **do**
2: **if** M_C are on the left of i_n **then**
3: $\mathbf{X}_{\text{sub}}$ = inplace-mat ($\underline{\mathbf{X}}$, M_C, i_n);
4: $\mathbf{Y}_{\text{sub}}$ = inplace-mat ($\underline{\mathbf{Y}}$, M_C, j);
// Matrix-matrix multiplication, using P_C threads
5: $\mathbf{Y}_{\text{sub}} = \mathbf{X}_{\text{sub}}\mathbf{U}'$, $\mathbf{U}'$ is the transpose of $\mathbf{U}$.
6: **else**
7: $\mathbf{X}_{\text{sub}}$ = inplace-mat ($\underline{\mathbf{X}}$, i_n, M_C);
8: $\mathbf{Y}_{\text{sub}}$ = inplace-mat ($\underline{\mathbf{Y}}$, j, M_C);
// Matrix-matrix multiplication, using P_C threads
9: $\mathbf{Y}_{\text{sub}} = \mathbf{U}\mathbf{X}_{\text{sub}}$
10: **end if**
11: **end parfor**
12: **return** $\underline{Y}$;

Algorithm 2: In-place Tensor-Times-Matrix Multiply (INTTM) algorithm to compute a mode-n product. "inplace-mat" means in-place building a sub-tensor (matrix) from initial full tensor, using modes for its row and column respectively.

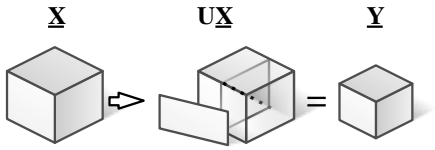


Figure 6: Structure of new InTtm computation.

In algorithm 2, we avoid physical reorganization of the $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$ tensors, and build a MM kernel accordingly using the chosen component modes (M_C). If M_C is assigned to the modes left of i_n, $\mathbf{X}_{\text{sub}}$ and $\mathbf{Y}_{\text{sub}}$ are in-place constructed with the size of $\prod_{i_c \in M_C} I_c \times I_n$ and $\prod_{i_c \in M_C} I_c \times J$ respectively. Mode-n is taken as the column of $\mathbf{X}_{\text{sub}}$ and $\mathbf{Y}_{\text{sub}}$, thus we use $\mathbf{Y}_{\text{sub}} = \mathbf{X}_{\text{sub}}\mathbf{U}'$ as the MM kernel. Otherwise, M_C is assigned to the modes to the right of i_n, mode-n is taken as the row of $\mathbf{X}_{\text{sub}}$ and $\mathbf{Y}_{\text{sub}}$. Thus we use $\mathbf{Y}_{\text{sub}} = \mathbf{U}\mathbf{X}_{\text{sub}}$ as the MM kernel. Note that sub-tensors (matrices) are not explicitly built, but implicitly referred to logically sub-tensors. Multi-threaded parallelism is additionally employed on both nested loops and the MM kernel, exposing two additional parameters. The parameter configuration will be described in the next section.

By avoiding an explicit data copy, the intensity $\widetilde{A}$ of INTTM algorithm will be,

$$\widetilde{A} \lesssim \frac{\hat{Q}}{\frac{\hat{Q}}{8\sqrt{Z}}} = 8\sqrt{Z} \approx A. \tag{6}$$

Our in-place INTTM algorithm improves the arithmetic intensity of tensor-times-matrix multiply, by eliminating the factor $1 + \frac{A}{m}$. The arithmetic intensity of INTTM is close to GEMM, so it has the potential to achieve comparable performance to GEMM. Furthermore, utilizing in-place operations

decreases storage space by approximately 50%.

The InTTM algorithm also presents new challenges. *Challenge 1:* InTTM does not have a natural static representation. As shown in algorithm 2, loop modes M_L and component modes M_C vary with the input tensor and mode-n. Because its performance might depend on the input, InTTM algorithm is a natural candidate for code-generation and auto-tuning. *Challenge 2:* InTTM algorithm may operate on inputs in a variety of shapes, as opposed to only square matrices. For instance, it would be common in InTTM in the case of a third-order tensor for two of the matrix dimensions to be relatively small compared to the third. Additionally, there might be large strides.

Despite these challenges, we still have opportunities for optimization. To this end, we build an input-adaptive framework that generates code given general input parameters. We also embed optimizations to determine the four parameter values M_L, M_C, P_L, and P_C in this framework, to generate an optimal tensor-times-matrix multiply code.

4.3 An Input Adaptive Framework

Our input-adaptive framework is shown in figure 7, illustrating the procedure of generating InTTM for a given tensor. There are three stages: input, parameter estimation, and loop generation, which generates the InTTM code. Input parameters include data layout, input tensor, leading mode, MM Benchmark, and the maximum supported number of threads. The parameter estimator predicts the optimal values of intermediate parameters, including loop order, M_L, M_C, P_L, and P_C. These intermediate parameters guide the generation of InTTM code. Within InTTM, either the Blis or MKL libraries will be called according to the parameter configuration.

We first illustrate parameter estimation, then explain the code generation process in the following sections.

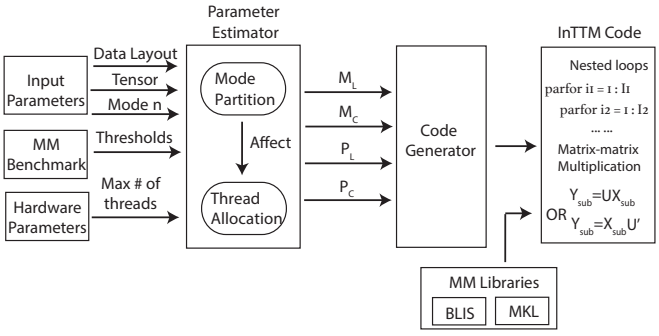


Figure 7: Input adaptive framework

4.3.1 Parameter Estimation

The two main aspects of parameter estimation are mode partitioning and thread allocation. Mode partitioning is the most important, and its decision influences thread allocation, so we state it first.

Mode partitioning. Apart from mode-n, all tensor modes are partitioned into two sets M_L and M_C, to generate nested loops and sub-tensors for inner matrix multiply. From figure 5, matrices in different sizes achieve very different performance numbers. In mode partitioning, we primarily decide M_C to maximize MM kernel's performance, and the rest modes are assigned to M_L.

Lemma 4.1 implies that there are two ways to choose contiguous modes, namely, from the modes either to the left or right of mode-n. We refer to these options as the *forward strategy* and *backward strategy*, respectively. In the forward strategy, the mode set is $M_C = \{m_2, \cdots, N\}$, where $n+1 \leq m_2 \leq N$; whereas in the backward strategy, the mode set $M_C = \{1, \cdots, m_1\}$, where $1 \leq m_1 \leq n-1$. If a tensor is stored in row-major pattern, the forward strategy generates a MM kernel with unit stride access in one dimension, while the backward strategy employs a general stride MM kernel. This means using forward strategy MM kernel can call the Blas, but the backward strategy would need general stride support, as provided by Blis. Different stride sizes and MM kernel implementations affect InTTM performance. As one would expect and experiments confirm, it is usually not possible for Blis operating on large general strides to achieve performance comparable to MKL when one dimension has unit stride. In the experiments below, we assume row-major layout, in which case we adopt the forward strategy. (One would use the backward strategy if the default layout were assumed to be column-major.)

After determining the strategy, the parameter m_1 (or m_2) in lemma 4.1 also needs to be determined. We introduce another parameter, *degree*, which specifies the number of component modes in M_C. Once *degree* is decided, so is m_1 (or m_2). For instance, take a three-dimensional tensor, $\underline{\mathbf{X}} \in \mathbb{R}^{100\times100\times100}$ and a matrix $\mathbf{U} \in \mathbb{R}^{16\times100}$, and suppose we wish to compute mode-1 product using the forward strategy. When $degree = 1$, $m_2 = 3$, $M_C = \{i_3\}$, $\mathbf{X}_{\text{sub}} \in \mathbb{R}^{100\times100}$. When $degree = 2$, $m_2 = 2$, $M_C = \{i_2, i_3\}$, $\mathbf{X}_{\text{sub}} \in \mathbb{R}^{100\times10000}$. Thus, different values of *degree* imply different sub-tensors and MM kernel sizes. As shown in figure 5, one would therefore expect performance to vary with different values of k and n, when m is fixed to a small value. Based on this observation, in our scheme we build a MM benchmark and use it to generate two thresholds, $MSTH$ and $MLTH$, which are used to determine the optimal *degree* and then M_C. To determine the thresholds, there are two steps. The first step is to fix k while varying n, since m is generally fixed when the tensors arise in some low-rank decomposition algorithm; and the second step is varying k.

Figure 8 shows MM performance on different values of n, when m and k are fixed. This figure shows a clear trend: after performance reaches a top-most value, as n increases, MM performance rebounds. We see a similar trend with other values of m and k. Matrices with unbalanced dimensions do not achieve high performance, in part because it is more difficult to apply a multilevel blocking strategy, as in Goto [11]. Put differently, when one dimension is relatively small, there may not actually be enough computation to hide data transfer time.

Figure 8 has a typical shape, which is some function $f(n)$ having a peak, $f_{\max}$. Consider a horizontal line, $\kappa f_{\max}$, for some fraction $\kappa = 0.8$, chosen empirically. The thresholds $MSTH$ and $MLTH$ correspond to the two red bars closest to but below the horizontal line. We set $MSTH$ ($MLTH$) as the storage size of the three matrices of the MM kernel with chosen n values. The second step is to calculate the average of each threshold over different values of k. We wish to find the MM kernel with the three matrices' storage size between the computed values of $MSTH$ and $MLTH$, which is relatively more likely to produce high Gemm performance. The

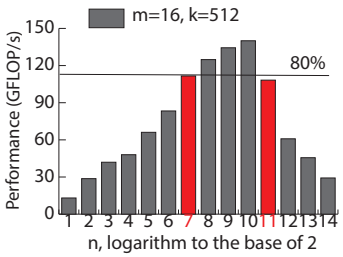


Figure 8: Performance variation of MM on different sizes of n, when $m = 16$, $k = 512$, using 4 threads.

size bounds of the MM kernel limit the *degree* parameter in our InTtm algorithm. From our experiments on a particular machine (with a Core i7 processor in our experiment), $MSTH$ is evaluated to 1.04 MB and $MLTH$ is evaluated to 7.04 MB.

The parameter *degree* is first initialized to 1, and its MM kernel's storage size is calculated. If it is smaller than $MSTH$, we increment *degree* and re-evaluate the storage size until we find the maximum MM kernel size between $MSTH$ and $MLTH$. Now, we use *degree* to partition modes to M_L and M_C.

From figure 7, the data layout also affects the partitioning process. Recall that the assumed data layout affects the choice of a forward (row-major) versus a backward (column-major) strategy. So, if $degree = p$, M_C is $\{N-p+1, \cdots, N\}$ (row-major) or $\{1, \cdots, p\}$ (column-major). This choice in turn means that the component modes should be chosen from the leading dimension, to guarantee unit stride in one dimension. The data layout also decides the order of M_L modes, where loop order is an increasing sequence of dimensions for row-major pattern.

Thread allocation. After determining M_C, we decide P_L and P_C according to the MM kernel size. From our tests, if the matrix is small, parallelizing the nested loops is more efficient than parallelizing the matrix multiply. For large matrices, the situation is the opposite. We use a threshold PTH for thread allocation, which is also shown as the storage size of the MM kernel. The difference is that PTH is determined from InTtm experiments, not from MM benchmark. If the size of MM kernel is smaller than PTH, we allocate more threads to nested loops; otherwise, more threads are allocated to MM kernel. The value of PTH is set to 800 KB in our tests. From our experiments, the highest performance is always achieved when using maximum threads on either nested loops or the MM kernel, so we only consider these two situations.

4.3.2 Code Generation

The code generation process consists of two pieces: generating nested loops and generating wrappers for the matrix multiply kernel. For each mode in set M_L, we build a for loop for it according to the mode order established by M_L. P_L is the number of threads allocated in nested loops. Code is generated in C++, using OpenMP with the collapse directive.

For the matrix multiply kernel, we build in-place sub-tensors $\mathbf{X}_{\text{sub}}$ and $\mathbf{Y}_{\text{sub}}$ using modes in M_C. According to the row and column strides, we choose between Blis [43] or Intel MKL libraries [17]. Thus, a complete InTtm code is generated according to the determined parameters.

5. EXPERIMENTS AND ANALYSIS

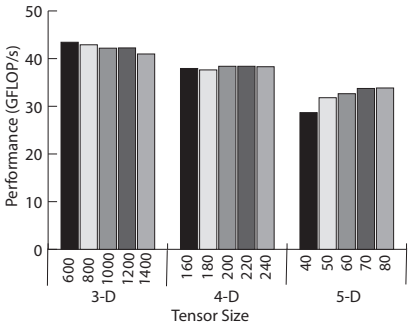
Our experimental evaluation focuses on three aspects of InTensLi: (a) assessing the absolute performance (GFLOP/s) of its generated implementations; (b) comparing the framework against widely used alternatives; and (c) verifying that its parameter tuning heuristics are effective. This evaluation is based on experiments carried out on the two platforms shown in table 2, one based on a Core i7-4770K processor and the other on a Xeon E7-4820. Our experiments employ 8 and 32 threads on the two platforms respectively, considering hyper-threading. The system based on the Xeon E7-4820 has a relatively large memory (512 GiB), allowing us to test a much larger range of (dense) tensor sizes than has been common in prior single-node studies. Note that all computations are performed in double-precision.

Table 2: Experimental Platforms Configuration

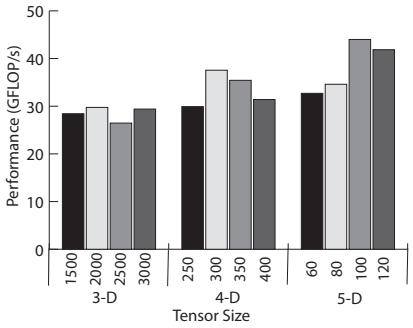
Parameters	Intel Core i7-4770K	Intel Xeon E7-4820
Microarchitecture	Haswell	Westmere
Frequency	3.5 GHz	2.0 GHz
# of physical cores	4	16
Hyper-threading	On	On
Peak GFLOP/s	224	128
Last-level cache	8 GiB	18 GiB
Memory size	32 GiB	512 GiB
Memory bandwidth	25.6 GB/s	34.2 GB/s
# of memory channels	2	4
Compiler	icc 15.0.2	icc 15.0.0

Basic benchmark. We first check that the InTensLi-generated InTtm delivers consistent performance at a variety of tensor dimensions and sizes. The results for a mode-2 product, as an example, appear in figure 9. We tested $\underline{\mathbf{X}} \times_2 \mathbf{U}$ where $\underline{\mathbf{X}}$ is an order-d tensor of size m in each dimension (total tensor size is m^d) and $\mathbf{U}$ is a $16 \times m$ matrix to agree with the low-rank property of tensor decomposition. We test $d \in \{3, 4, 5\}$ at various values of m, shown along the x-axis, chosen so that the largest m still permits m^d to fit into main memory; the y-axis shows performance in GFLOP/s. On the Core i7, our InTtm achieves over 40 GFLOP/s on 3rd-order tensors, with performance tending to steadily decrease or remain flat with increasing size and order. At higher orders, dimension size decreases in order to fit into memory, which reduces the inner Gemm performance as we would expect from the observations of § 3. By contrast, performance trends on the Xeon E7 platform differ from those on the Core i7. In particular, 3rd-order tensors show the worst performance, compared to higher-order tensors. This stems in part from worse Intel MKL performance on the Xeon E7—it achieves only 51 GFLOP/s on a square Gemm with operands of size 1000×1000, compared to 154 GFLOP/s on the Core i7. However, it also happens that, on the Xeon E7, the multithreading within MKL does not appear to benefit Gemm performance much. Our ability to achieve higher performance with higher orders on that platform comes mainly from our use of coarse-grained outer-loop parallelization.

On both platforms, InTensLi-generated InTtm does not deliver performance that compares well with the peak performance shown in table 2. The main reason is, as noted



(a) Intel Core i7-4770K



(b) Intel Xeon E7-4820

Figure 9: Performance of InTensLi-generated InTtm algorithm for mode-2 product on 3rd, 4th, and 5th-order tensors. Each bar represents the performance of a specific tensor size.

in § 3 and figure 5, GEMM performance differs from peak for rectangular shapes. Such shapes can arise in INTENSLI-generated TTM code, since it considers a variety of ways to aggregate and partition loops.

Comparison to other tools. We compare INTENSLI-generated INTTM code against the equivalent implementations in the TENSOR TOOLBOX and CTF. Note that the TENSOR TOOLBOX and CTF implementations use algorithm 1. TENSOR TOOLBOX is designed to be MATLAB-callable, but under-the-hood uses multithreading and can link against a highly-tuned BLAS. CTF is implemented in C++, with OpenMP parallelization and BLAS calls where possible. In addition, it is useful to also compare against GEMM using a matricized tensor (line 4 of algorithm 1) but *ignoring* any reorganization time and other overhead. This measurement of just GEMM provides an estimate of the performance we might expect to achieve. TENSOR TOOLBOX, CTF, GEMM, and our INTTM are all linked to MKL library for GEMM calls.

The results appear in figure 10. Because TENSOR TOOLBOX and CTF have larger memory requirements than the INTTM, the tensor sizes selected for figure 10 are smaller than for figure 9. In figure 10, the leftmost bar is INTENSLI-generated INTTM, which achieves the highest performance among the four. The performance of TENSOR TOOLBOX and CTF is relatively low, at about 10 GFLOP/s and 3 GFLOP/s, respectively. Our INTTM gets about 4× and 13× speedups compared to TENSOR TOOLBOX and CTF. The main reason is that TENSOR TOOLBOX and CTF incur overheads from explicit copies. The rightmost bar is GEMM-only performance.

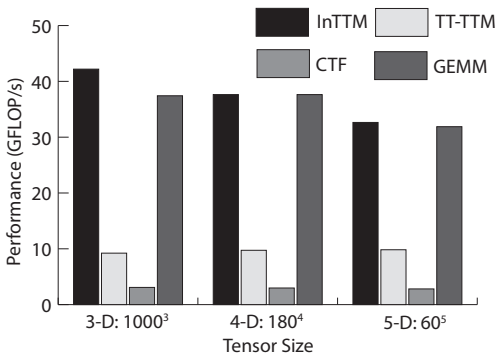


Figure 10: Performance comparison among InTensLi-generated InTtm, Tensor Toolbox (TT-TTM), Cyclops Tensor Framework (Ctf), and Gemm on 3rd, 4th, and 5th-order tensors of mode-2 product.

Our INTTM matches it, and can even outperform it since the INTENSLI framework considers a larger space of implementation options than what is possible through algorithm 1.

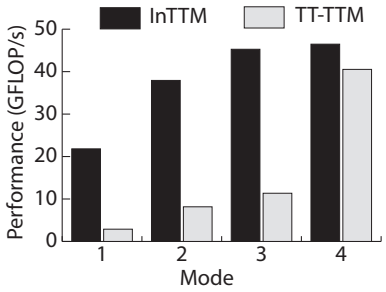


Figure 11: Performance behavior of InTensLi-generated InTtm against Tensor Toolbox (TT-TTM) for different mode products on a $160 \times 160 \times 160 \times 160$ **4th-order tensor.**

We also compare against just TENSOR TOOLBOX for varying mode-n computations in figure 11, on a 4th-order tensor. [4] X-axis shows the modes corresponding to our INTTM algorithm. The TENSOR TOOLBOX performance varies significantly among different modes, ranging from 3 GFLOP/s to 40 GFLOP/s. By contrast, the INTENSLI-TTM reduces this performance variability with changing mode. This result shows the benefit of specializing the partitioning and iteration-ordering strategy, as INTENSLI does automatically.

Parameter selection. The last major aspect of the INTENSLI framework is selecting good parameters. Recall that during the mode partitioning process, a GEMM benchmark is used to determine two thresholds, $MSTH$ and $MLTH$. In figure 12, we compare the results of using INTENSLI's heuristics to choose these parameters against an exhaustive

[4]The TENSOR TOOLBOX uses a column-major ordering, whereas INTENSLI uses a row-major ordering. As such, for a order-d tensor, we compare TENSOR TOOLBOX's mode-n product against our mode-$(d-n+1)$ product, i.e., their mode-1 against our mode-4 product, their mode-2 against our mode-3 product, and so on. In this way, the storage pattern effect is eliminated.

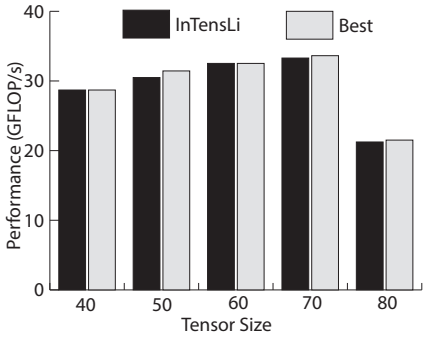


Figure 12: Comparison between the performance with predicted configuration and the actual highest performance on 5th-order tensors of mode-1 product.

search, in the case of a mode-1 product on a 5th-order tensor. The black bars show the performance of the configuration chosen automatically by InTensLi, and the gray bars show the performance of the configuration chosen by exhaustive search. For this particular input, there are 16 possible configurations, among which InTensLi's heuristics select just 1. As figure 12 shows, InTensLi makes near-optimal choices.

From these experiments, InTensLi can easily be integrated in tensor decompositions, such as Tucker decomposition for low-order tensors and hierarchical Tucker for high-dimensional tensors. Such applications can benefit from the performance and space-efficiency of InTensLi-generated code.

6. RELATED WORK

One of the most widely-used tensor implementations is the Tensor Toolbox. Its optimized Ttm kernel has been the baseline comparison for our experiments. This implementation utilizes an algorithm (METTM) [22] that alleviates the intermediate memory-blowup problem. By carefully choosing which modes to compute with finer granularity, the intermediate data remains within working memory. Tensor Toolbox suffers from excessive data copy according to our experiments, which motivates our in-place approach.

Cyclops Tensor Framework (Ctf) [40] provides another baseline implementation. Ctf is a recent HPC implementation with two levels of parallelism (OpenMP and MPI) which focuses on communication reduction for symmetric tensor contractions that arise frequently in quantum chemistry. Ttm is a specific instance of tensor contraction. CTF distributes tensors via a slice-representation. Another implementation that specializes on contraction is the Tensor Contraction Engine (TCE) [5], a mature implementation that focuses on synthesizing code and dynamically maintains load balance on distributed systems. TCE also builds a model to choose optimal data layout, while we choose from different matrix shapes. InTensLi-generated InTtm can serve as a single-node implementation for a distributed version.

As discussed in table 1, there are many different ways to think about and represent the same tensor operation, and there has been a recent flurry of work on rethinking tensor representations to make for more efficient, scalable decompositions.

The Matricized Tensor Times Khatri-Rao Product (MTTKRP) is an essential step of CANDECOMP/PARAFAC (CP) Decomposition, and differs from general Ttm in that the matrix is the result of the Khatri-Rao product of two matrices. N.Ravindran, et. al created an in-place tensor-matrix product for MTTKRP [33], but their implementation operates on the 'slice' representation of the tensor. Our work takes advantage of a more general subtensor representation, and in particular its opportunities for performance tuning.

A number of sparse implementations have been proposed as well: GigaTensor [20] restructures MTTKRP as a series of Hadamard products in the MapReduce framework, which increases parallelism at the cost of more work. This general approach has had success scaling to very large tensors [18]. DFacTo [9] restructures the operation as a series of sparse distributed matrix-vector multiplies, and SPLATT [38] computes along tensor slices while exploiting sparsity patterns to improve cache utilization in shared-memory. Though our implementation is dense, the algorithm can be applied to sparse tensors provided a sparse matrix multiply kernel is provided (SpGEMM).

Baskaran et al. [8] implement a data structure approach to improving the performance of tensor decompositions, proposing a sparse storage format that can be organized along modes in such a way that data reuse is increased for the Tucker Decomposition. In contrast, our approach avoids the overhead of maintaining a separate data structure and can use native, optimized multiply operations.

7. CONCLUSIONS

This paper's key finding is that a mode-n product, or Ttm operation, can be performed efficiently in-place and tuned automatically to the order and dimensions of the input tensor. Our InTensLi framework can serve as a template for other primitives and tensor products, a few of which appear in § 6. Although we focused on improving single-node performance, including exploiting shared memory parallelism and reducing Ttm bandwidth requirements, this building block can be used as a "drop-in" replacement for the intra-node compute component of distributed memory implementations, which we will pursue as part of our future work.

There are several additional avenues for future work. One is to show the impact of our performance improvements in the context of higher-level decomposition algorithms, such as Tucker, hierarchical Tucker, or tensor trains, among others [12, 14, 30, 42]. The case of dense tensors has numerous scientific applications, including, for instance, time series analysis for molecular dynamics, which we are pursuing [32]. Beyond the dense case, *sparse* tensors primitives pose a number of new challenges, which include efficient data structure design and iteration. The sparse case is as an especially important class, with many emerging applications in data analysis and mining [10].

Acknowledgments

This material is based upon work supported by the U.S. National Science Foundation (NSF) Award Number 1339745, and Award Number 1337177. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF.

References

[1] An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.

[2] E. Acar, C. Aykut-Bingol, H. Bingol, R. Bro, and B. Yener. Multiway analysis of epilepsy tensors. *Bioinformatics*, 23(13):i10–i18, 2007.

[3] E. Acar, S. A. Camtepe, M. S. Krishnamoorthy, and B. Yener. Modeling and multiway analysis of chatroom tensors. In *Intelligence and Security Informatics*, pages 256–268. Springer, 2005.

[4] E. Acar, R. J. Harrison, F. Olken, O. Alter, M. Helal, L. Omberg, B. Bader, A. Kennedy, H. Park, Z. Bai, D. Kim, R. Plemmons, G. Beylkin, T. Kolda, S. Ragnarsson, L. Delathauwer, J. Langou, S. P. Ponnapalli, I. Dhillon, L.-h. Lim, J. R. Ramanujam, C. Ding, M. Mahoney, J. Raynolds, L. EldÃl'n, C. Martin, P. Regalia, P. Drineas, M. Mohlenkamp, C. Faloutsos, J. Morton, B. Savas, S. Friedland, L. Mullin, and C. Van Loan. Future directions in tensor-based computation and modeling, 2009.

[5] A. Auer and etc. Automatic code generation for many-body electronic structure methods: the tensor contrac. *Molecular Physics*, 104(2):211–228, 2006.

[6] B. W. Bader, T. G. Kolda, et al. Matlab tensor toolbox version 2.5. Available from http://www.sandia.gov/ tgkolda/TensorToolbox/, January 2012.

[7] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:pp. 1–155, 2014.

[8] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. Efficient and scalable computations with sparse tensors. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–6, Sept 2012.

[9] J. H. Choi and S. Vishwanathan. Dfacto: Distributed factorization of tensors. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1296–1304. Curran Associates, Inc., 2014.

[10] A. Cichocki. Era of big data processing: A new approach via tensor networks and tensor decompositions. *CoRR*, abs/1403.2048, 2014.

[11] K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[12] L. Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM J. Matrix Anal. Appl.*, 31(4):2029–2054, May 2010.

[13] L. Grasedyck, D. Kressner, and C. Tobler. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen*, 36(1):53–78, 2013.

[14] R. A. Harshman. Foundations of the parafac procedure: models and conditions for an" explanatory" multimodal factor analysis. 1970.

[15] J. C. Ho, J. Ghosh, and J. Sun. Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 115–124, New York, NY, USA, 2014. ACM.

[16] R. W. Hockney and I. J. Curington. $f_{\frac{1}{2}}$: A parameter to characterize memory and communication bottlenecks. *Parallel Computing*, 10:277–286, 1989.

[17] Intel. Math kernel library. http://developer.intel.com/software/products/mkl/.

[18] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos. Haten2: Billion-scale tensor decompositions. In *ICDE*, 2015.

[19] M. Jiang, P. Cui, F. Wang, X. Xu, W. Zhu, and S. Yang. Fema: Flexible evolutionary multi-faceted analysis for dynamic behavioral pattern discovery. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1186–1195, New York, NY, USA, 2014. ACM.

[20] U. Kang, E. E. Papalexakis, A. Harpale, and C. Faloutsos. Gigatensor: scaling tensor analysis up by 100 times - algorithms and discoveries. In *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, pages 316–324, 2012.

[21] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[22] T. G. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, pages 363–372, Washington, DC, USA, 2008. IEEE Computer Society.

[23] C.-F. V. Latchoumane, F.-B. Vialatte, J. Solé-Casals, M. Maurice, S. R. Wimalaratna, N. Hudson, J. Jeong, and A. Cichocki. Multiway array decomposition analysis of eegs in alzheimer's disease. *Journal of neuroscience methods*, 207(1):41–50, 2012.

[24] L. D. Lathauwer and J. Vandewalle. Dimensionality reduction in higher-order signal processing and rank-(r1,r2,...,rn) reduction in multilinear algebra. *Linear Algebra and its Applications*, 391(0):31 – 55, 2004. Special Issue on Linear Algebra in Signal and Image Processing.

[25] J. Li, G. Tan, M. Chen, and N. Sun. Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 117–126, New York, NY, USA, 2013. ACM.

[26] Y. Matsubara, Y. Sakurai, W. G. van Panhuis, and C. Faloutsos. Funnel: Automatic mining of spatially coevolving epidemics. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 105–114, New York, NY, USA, 2014. ACM.

[27] J. Mocks. Topographic components model for event-related potentials and some biophysical considerations. *Biomedical Engineering, IEEE Transactions on*, 35(6):482–484, June 1988.

[28] M. Morup, L. K. Hansen, C. S. Herrmann, J. Parnas, and S. M. Arnfred. Parallel factor analysis as an exploratory tool for wavelet transformed event-related {EEG}. *NeuroImage*, 29(3):938 – 947, 2006.

[29] J. Nagy and M. Kilmer. Kronecker product approximation for preconditioning in three-dimensional imaging applications. *Image Processing, IEEE Transactions on*, 15(3):604–613, March 2006.

[30] I. V. Oseledets. Tensor-train decomposition. *SIAM J. Scientific Computing*, 33(5):2295–2317, 2011.

[31] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropolous. ParCube: Sparse parallelizable tensor decompositions. In *Proceedings of the 2012 European Conference on Machine Learning Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, pages pp. 521–536, Bristol, United Kingdom, 2012.

[32] A. Ramanathan, P. K. Agarwal, M. Kurnikova, and C. J. Langmead. *An online approach for mining collective behaviors from molecular dynamics simulations*, volume LNCS 5541, pages pp. 138–154. 2009.

[33] N. Ravindran, N. D. Sidiropoulos, S. Smith, and G. Karypis. Memory-efficient parallel computation of tensor and matrix products for big tensor decompositions. *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, 2014.

[34] B. Savas and L. Eldén. Handwritten digit classification using higher order singular value decomposition. *Pattern recognition*, 40(3):993–1003, 2007.

[35] A. Shashua and A. Levin. Linear image coding for regression and classification using the tensor-rank principle. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–42–I–49 vol.1, 2001.

[36] N. Sidiropoulos, R. Bro, and G. Giannakis. Parallel factor analysis in sensor array processing. *Signal Processing, IEEE Transactions on*, 48(8):2377–2388, Aug 2000.

[37] N. Sidiropoulos, G. Giannakis, and R. Bro. Blind parafac receivers for ds-cdma systems. *Signal Processing, IEEE Transactions on*, 48(3):810–823, Mar 2000.

[38] S. Smith, N. Ravindran, N. Sidiropoulos, and G. Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, IPDPS, 2015.

[39] E. Solomonik, J. Demmel, and T. Hoefler. Communication lower bounds for tensor contraction algorithms. Technical report, ETH Zürich, 2015.

[40] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel. Cyclops tensor framework: reducing communication and eliminating load imbalance in massively parallel contractions. Technical Report UCB/EECS-2012-210, EECS Department, University of California, Berkeley, Nov 2012.

[41] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 374–383. ACM, 2006.

[42] L. R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

[43] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 2013.

[44] M. A. O. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *Computer Vision-ECCV 2002*, pages 447–460. Springer, 2002.

[45] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.