High performance tensor-vector multiplies on shared memory systems

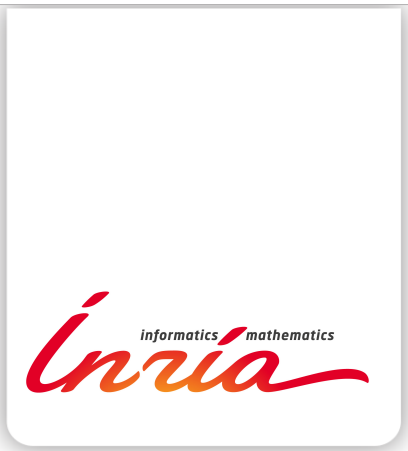
Filip Pawlowski, Bora Uçar, Albert-Jan Yzelman

High performance tensor–vector multiplies on shared-memory systems

Filip Pawłowski, Bora Uçar, Albert-Jan Yzelman

RESEARCH REPORT
N° 9274
May 2019
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9274--FR+ENG



High performance tensor–vector multiplies on shared-memory systems

Filip Pawłowski*, Bora Uçar†, Albert-Jan Yzelman‡

Project-Team ROMA

Research Report n° 9274 — May 2019 — 20 pages

Abstract: Tensor–vector multiplication is one of the core components in tensor computations. We have recently investigated high performance, single core implementation of this bandwidth-bound operation. Here, we investigate its efficient, shared-memory implementations. Upon carefully analyzing the design space, we implement a number of alternatives using OpenMP and compare them experimentally. Experimental results on up to 8 socket systems show near peak performance for the proposed algorithms.

Key-words: tensors, tensor–vector multiplication, shared-memory parallel machines

* Huawei Technologies France and ENS Lyon
† CNRS and LIP (UMR5668 Université de Lyon - CNRS - ENS Lyon - Inria - UCBL 1), 46, allée d'Italie, ENS Lyon, Lyon F-69364, France.
‡ Huawei Technologies France

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Multiplication tenseur–vecteur haute performance sur des machines à memoire partagée

Résumé : La multiplication tenseur–vecteur est l'un des composants essentiels des calculs de tenseurs. Nous avons récemment étudié cette opération, qui consomme la bande passante, sur une plateforme séquentielle. Dans ce travail, nous étudions des algorithmes efficaces pour effectuer cette opération sur des machines à mémoire partagée. Après avoir soigneusement analysé les différentes alternatives, nous mettons en œuvre plusieurs d'entre elles en utilisant OpenMP, et nous les comparons expérimentalement. Les résultats expérimentaux sur un à huit systèmes de sockets montrent une performance quasi maximale pour les algorithmes proposés.

Mots-clés : tenseur, multiplication tenseur-vecteur, machines parallèles à mémoire partagée

1 Introduction

Tensor–vector multiply (TVM) operation, along with its higher level analogues tensor–matrix (TMM) and tensor–tensor multiplies (TTM) are the building blocks of many algorithms [1]. These operations are applied to a given mode (or dimension), or to given modes (in the case of TTM). Among these, TVM is the most bandwidth-bound. Recently, we have investigated this operation on single core systems, and proposed data structures and algorithms to achieve high performance and mode-oblivious behavior [9]. While high performance is a common term in the close by area of matrix computations, mode-obliviousness is mostly related to tensor computations. It requires that a given algorithm for a core operation (e.g., TVM) should have more or less the same performance no matter which mode it is applied to. In matrix terms, this corresponds to having the same performance in computing matrix–vector and matrix–transpose–vector multiplies. Our aim in this work is to develop high performance and mode oblivious parallel TVM algorithms on shared-memory systems.

Let $\mathcal{A}$ be a tensor with d modes, or for our purposes in this paper, a d-dimensional array. The k-mode tensor–vector multiplication produces another tensor whose kth mode is of size one. More formally, for $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ and $\mathbf{x} \in \mathbb{R}^{n_k}$, the k-mode TVM operation $\mathcal{Y} = \mathcal{A} \times_k \mathbf{x}$ is defined as

$$y_{i_1,\ldots,i_{k-1},1,i_{k+1},\ldots,i_d} = \sum_{i_k=1}^{n_k} a_{i_1,\ldots,i_{k-1},i_k,i_{k+1},\ldots,i_d} x_{i_k},$$

for all $i_j \in \{1,\ldots,n_j\}$ with $j \in \{1,\ldots,d\}$, where $y_{i_1,\ldots,i_{k-1},1,i_{k+1},\ldots,i_d}$ is an element of $\mathcal{Y}$, and $a_{i_1,\ldots,i_{k-1},i_k,i_{k+1},\ldots,i_d}$ is an element of $\mathcal{A}$. The output tensor $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_{k-1} \times 1 \times n_{k+1} \times \cdots \times n_d}$ is $d-1$-dimensional. That is why one can also state that the k-mode TVM contracts a d-dimensional tensor along mode k and forms a $d-1$-dimensional tensor. Note that the operation $\mathcal{Y} = \mathcal{A} \times_k \mathbf{x}$ is special from the computational point of view: the size of one of its inputs, $\mathcal{A}$, is much greater than the other input, $\mathbf{x}$. Let $n = \prod_{i=1}^{d} n_i$. Then, a k-mode TVM performs $2n$ flops on $n + n/n_k + n_k$ data elements, and thus has arithmetic intensity of $\frac{2n}{n+n/n_k+n_k}$ flop per word, which is between 1 and 2. This amounts to a heavily bandwidth-bound computation even for sequential execution [9]. The multi-threaded case is even more challenging, as cores on a single socket compete for the local memory bandwidth.

We proposed [9] a blocking approach for obtaining efficient, mode-oblivious tensor computations by investigating the case of tensor–vector multiplication. Ballard et al. [2] investigate the communication requirements of a well-known operation called MTTKRP and discuss a blocking approach. MTTKRP is usually formulated by matrix–matrix multiplication using BLAS libraries. Earlier approaches to this and related operations unfold the tensor (reorganize the whole tensor in the memory), and carry out the overall operation using a single matrix–matrix multiplication [5]. Li et al. [6] instead propose a parallel loop-based algorithm: a loop of the BLAS3 kernels, which operate in-place on parts of the tensor such that no unfold is required. Kjolstad et al. [4] propose The Tensor Algebra Compiler (taco) for tensor computations. Given a tensor algebraic expression, mixing tensors of different dimensions and storages (sparse and dense), taco generates code for different modes of a tensor according to the operands of the expression. Tensor–tensor multiplication, or contraction, has received considerable attention. This operation is the most general form of the multiplication operation in (multi)linear algebra. CTF [10], TBLIS [7], and GETT [11] are recent libraries carrying out this operation based on principles and lessons learned from high performance matrix–matrix multiplication. Apart from not explicitly considering TVM, they do not adapt the

tensor layout. As a consequence, they all require transpositions, one way or another. Our *TVM* routines address a special case of *TMM*, which is a special case of *TTM*, based on our earlier work [9].

We list the notation in Section 2, and provide a background on blocking algorithms we proposed earlier for sequential high performance. Section 3 contains *TVM* algorithms whose analyses are presented in Section 4. Section 5 contains experiments on up to 8-socket 120 core systems.

2 Notation and background

2.1 Notation

We use mostly the standard notation [5] (the full list of symbols is given in Table 1 in the technical report [**?**]). $\mathcal{A}$ is an order-d, or a d-dimensional tensor. $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ has size n_k in mode $k \in \{1, \ldots, d\}$. $\mathcal{Y}$ is a $(d-1)$-dimensional tensor obtained by multiplying $\mathcal{A}$ along a given mode k with a suitably sized vector $\mathbf{x}$. Matrices are represented using boldface capital letters; vectors using boldface lowercase letters; and elements in them are represented by lowercase letters with subscripts for each dimension. When a subtensor, matrix, vector, or an element of a higher order object is referred, we retain the name of the parent object. For example $a_{i,j,k}$ is an element of the tensor $\mathcal{A}$. We use Matlab column notation for denoting all indices in a mode. For $k \in \{1, \ldots, d\}$, we use $I_k = \{1, \ldots, n_k\}$ to denote the index set for the mode k. We also use $n = \Pi_{i=1}^{d} n_i$ to refer to the total number of elements in $\mathcal{A}$. Likewise, $I = I_1 \times I_2 \times \cdots \times I_d$ is the Cartesian product of all index sets, whose elements are marked with boldface letters $\mathbf{i}$ and $\mathbf{j}$. For example, $a_{\mathbf{i}}$ is an element of $\mathcal{A}$ whose indices are $\mathbf{i} = i_1, \ldots, i_d$. A mode-$k$ fiber $\mathbf{a}_{i_1,\ldots,i_{k-1},:,i_{k+1},\ldots,i_d}$ in a tensor is obtained by fixing the indices in all modes except mode k. A hyperslice is obtained by fixing one of the indices, and varying all others. In third order tensors, a hyperslice become a slice, and therefore, a matrix. For example, $\mathbf{A}_{i,:,:}$ is the ith mode-1 slice of $\mathcal{A}$.

The total number of threads an algorithm employs is p, which need not be equal to the number of cores a given machine holds; it can be less when considering strong scalability, and it can be more when exploring the use of hyperthreads. Let $P = \{1, \ldots, p\}$ be the set of all possible thread IDs.

All symbols are summarized in Table 1.

2.2 Sequential *TVM* and dense tensor memory layouts

We focus on shared-memory parallel algorithms for computing a k-mode *TVM*. We parallelize the *TVM* by distributing the input tensor between the physical cores of a shared-memory machine, while adopting the tensor layouts and *TVM* kernels from our earlier work [9], summarized below.

A layout ρ maps tensor elements onto an array of size $n = \Pi_{i=1}^{d} n_i$. Most commonly, dense tensors are stored as multidimensional arrays. For instance, a matrix can be stored in two different ways (row-major and column-major), while d-dimensional tensors can be stored in $d!$ different ways, giving $d!$ unfolds [5]. Let $\rho_\pi(\mathcal{A})$ be a layout, and π an ordering (permutation) of $(1, \ldots, d)$ such that

$$\rho_\pi(\mathcal{A}) : (i_1, \ldots, i_d) \mapsto \sum_{k=1}^{d} \left((i_{\pi_k} - 1) \prod_{j=k+1}^{d} n_{\pi_j} \right) + 1 \,,$$

$\mathcal{A}, \mathcal{Y}$	An input and output tensor, respectively
$\mathbf{x}$	An input vector
d	The order of $\mathcal{A}$ and one plus the order of $\mathcal{Y}$
n_i	The size of $\mathcal{A}$ in the ith dimension
n	The number of elements in $\mathcal{A}$
I_i	The index set corresponding to n_i
I	The Cartesian product of all I_i
$\mathbf{i}$ and $\mathbf{j}$	Members of I
k	The mode of a TVM computation
b	Individual block size of tensors blocked using hypercubes
p_s	The number of sockets
p_t	The number of threads per socket
p	The total number of threads $p_\mathrm{s}p_\mathrm{t}$
s	The ID of a given thread
P	The set of all possible thread IDs
π	Any distribution of $\mathcal{A}$
$\pi_{1\mathrm{D}}$	A 1D block distribution
$b_{1\mathrm{D}}$	The block size of a load-balanced 1D block distribution
ρ_π	A unfold layout for storing a tensor
ρ_Z	A Morton order layout for storing a tensor
$\rho_Z\rho_\pi$	Blocked tensor layout with a Morton order on blocks
m_s	The number of fibers in each slice under a 1D distribution
$\mathcal{A}_s, \mathcal{Y}_s$	Thread-local versions of $\mathcal{A}, \mathcal{Y}$
M_s	Memory requirement at thread s
S	Number of barriers required
$U_{s,i}$	Intra-socket data movement for thread s in phase i
$V_{s,i}$	Inter-socket data movement for thread s in phase i
$W_{s,i}$	Work (in flops) at thread s in phase i
g, h	Intra- and inter-socket throughput (seconds per word)
r	Thread compute speed (seconds per flop)
L	The time for a barrier to complete (in seconds)
T_seq	The best sequential time
$T(n,p)$	The time taken for parallel execution on n elements and p threads
$O(n,p)$	The overhead of a parallel algorithm
$E(n,p)$	The efficiency of a parallel algorithm

Table 1 – Notation used throughout the paper

with the convention that $\prod_{j=k+1}^{d} \cdot = 1$ for $k = d$. The regularity of this layout allows such tensors be processed using BLAS in a loop without explicit tensor unfolds. Let $\rho_Z(\mathcal{A})$ be a Morton layout defined by the space-filling Morton order [8]. The Morton order is defined recursively, where at every step the covered space is subdivided into two within every dimension; for 2D planar areas this creates four cells, while for 3D it creates eight cells. In every two dimensions the order between cells is given by a (possibly rotated) Z-shape. Let w be the number of bits used to represent a single coordinate, and let $i_k = (l_1^k \dots l_w^k)_2$ for $k \in \{1, \dots, d\}$ be the bit representation of each coordinate. The Morton order in d dimensions $\rho_Z(\mathcal{A})$ can then be defined as

$$\rho_Z(\mathcal{A}) : (i_1, \dots, i_d) \mapsto (l_1^1 l_1^2 \dots l_1^d l_2^1 l_2^2 \dots l_2^d \dots l_w^1 l_w^2 \dots l_w^d)_2 \,.$$

Such layout improves performance on systems with multi-level caches due to the locality preserving properties of the Morton order. However, $\rho_Z(\mathcal{A})$ is an irregular layout, and thus unsuitable for processing with BLAS routines.

Blocking is a well-known technique for improving data locality. A blocked tensor consists of blocks $\mathcal{A}_j \in \mathbb{R}^{b_1 \times \dots \times b_d}$, where $j \in \{1, \dots, \prod_{i=1}^{d} a_i\}$, and $n_k = a_k b_k$ for all modes k. We previously introduced a $\rho_Z \rho_\pi$ blocked layout which organizes elements into blocks, and uses ρ_Z to order the blocks in memory, and ρ_π to order the elements in individual blocks [9]. By using the regular layout at the lower level, we can use BLAS routines for processing the individual blocks, while benefiting from the properties of the Morton order (increased data reuse between blocks, and mode-oblivious performance). The detailed description of sequential algorithms using these layouts can be found in our earlier work [9].

3 Shared-memory parallel *TVM* algorithms

3.1 Shared-memory parallelization

We assume a shared-memory architecture consisting of p_s connected processors. Each processor supports running p_t threads for a total of $p = p_s p_t$ threads. Each processor has local memory which can be accessed faster than remote memory areas. We assume threads taking part in a parallel *TVM* computation are *pinned* to a specific core, meaning that threads will not move from one core to another while a *TVM* is executed. A pinned thread has a notion of local memory: namely, all addresses that are mapped to the memory controlled by the processor the thread is pinned to. This gives rise to two distinct modes of use for shared memory areas: the *explicit* versus *interleaved* modes. If a thread allocates, initialises, and remains the only thread using this memory area, we dub its use explicit. In contrast, if the memory pages associated with an area cycle through all available memories, then the use is called interleaved. If a memory area is accessed by all threads in a uniformly random fashion, then it is advisable to interleave to achieve high throughput.

We will consider both parallelizations of for-loops as well as parallelizations following the Single Program, Multiple Data (SPMD) paradigm. In the former, we identify a for-loop where each iterant can be processed concurrently without causing race conditions. Such a for-loop can either be cut *statically* or *dynamically*; the former cuts a loop of size n in exactly p parts and has each thread execute a unique part of the loop, while the latter typically employs a form of work stealing to assign parts of the loop to threads. In both cases, we assume that one does not explicitly control which thread will execute which part of the loop.

At the finest level, a distribution of an order-d tensor of size $n_1 \times \cdots \times n_d$ over p threads is given by a map $\pi : I \to \{1, \ldots, p\}$. Let $\pi_{1\mathrm{D}}$ be a regular 1D *block distribution* such that

$$\pi_{1\mathrm{D}}(\mathcal{A}) : (i_1, i_2, \ldots, i_d) \mapsto \lfloor (i_1 - 1)/b_{1\mathrm{D}} \rfloor + 1,$$

where *block size* $b_{1\mathrm{D}} = \lceil n_1/p \rceil$ refers to the number of hyperslices. Let $m_s = |\pi_{1\mathrm{D}}^{-1}(s)|$ count the number of elements local to thread s. We demand that a 1D distribution be *load-balanced*,

$$\max_{s \in P} m_s - \min_{s \in P} m_s \leq n/n_1.$$

The choices to distribute over the first mode and to use a block distribution are without loss of generality. The dimensions of $\mathcal{A}$ and their fibers could be permuted to fit any other load-balanced 1D distribution. For smaller sizes, the dimensions could be reordered, or fewer threads could be used.

3.2 Baseline: loopedBLAS

We assume $\mathcal{A}$ and $\mathcal{Y}$ have the default unfold layout. The *TVM* operation could naively be written using d nested for-loops, where the outermost loop that does not equal the mode k of the *TVM* is executed concurrently using OpenMP; such code is generated by taco. For a better performing parallel baseline, however, we observe that the $d - k$ inner for-loops correspond to a dense matrix–vector multiplication if $k < d$; we can thus write the parallel *TVM* as a loop over BLAS-2 calls, and use highly optimized libraries for their execution. For $k = d$, the naively nested for-loops actually correspond to a dense matrix–transpose–vector multiplication, which is a standard BLAS-2 call as well. Note that the matrices involved with these BLAS-2 calls generally are rectangular.

We execute the loop over the BLAS-2 calls in parallel using OpenMP. For $k = d$, and for smaller tensors, this may not expose enough parallelism to make use of all available threads; we use any such left-over threads to parallelize the BLAS-2 calls themselves, while taking care that threads collaborating on the same BLAS-2 call are pinned close to each other to exploit shared caches as much as possible. Since all threads access both the input tensor and input vector, and since it cannot be predicted which thread accesses which part of the output tensor, all memory areas corresponding to $\mathcal{A}$, $\mathcal{Y}$, and $\mathbf{x}$ must be interleaved. We refer to the described algorithm as *loopedBLAS*, which for $p = 1$ is equivalent to *tvLooped* in our earlier work [9].

3.3 Proposed 1D *TVM* algorithms

We explore a family of algorithms assuming the $\pi_{1\mathrm{D}}$ distribution of the input and output tensors, thus resulting in p disjoint input tensors $\mathcal{A}_s$ and p disjoint output tensors $\mathcal{Y}_s$ where each of their unions correspond to $\mathcal{A}$ and $\mathcal{Y}$, respectively. For all but $k = 1$, a parallel *TVM* amounts to a thread-local call to a sequential *TVM* computing $\mathcal{Y}_s = \mathcal{A}_s \times_k \mathbf{x}$; each thread reads from its own part of $\mathcal{A}$ while writing to its own part of $\mathcal{Y}$. We may thus employ the $\rho_Z \rho_\pi$ layout for $\mathcal{A}_s$ and $\mathcal{Y}_s$ and use its high-performance sequential mode-oblivious kernel [9]; here, $\mathbf{x}$ is allocated interleaved while $\mathcal{A}_s$ and $\mathcal{Y}_s$ are explicit. The global tensors $\mathcal{A}$ and $\mathcal{Y}$ are never materialized in shared-memory—only their distributed variants are required. We expect the explicit allocation of these two largest data entities involved with the *TVM* computation to induce much better parallel efficiency compared to the *loopedBLAS* baseline where all data is interleaved.

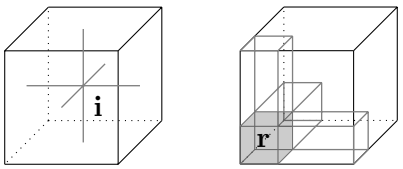


Figure 1 – Illustrations of elements in $J_{\mathbf{i}}$, indicated via thick gray lines, for an arbitrarily chosen $\mathbf{i}$ depicted by a filled dot (left), and for a cube of r elements $\mathbf{i}$ (right).

For $k = 1$, the output tensor $\mathcal{Y}$ cannot be distributed. We define that $\mathcal{Y}$ is then instead subject to a 1D block distribution over mode 2, and assume $n_2 \geq p$. Since the distributions of $\mathcal{A}$ and $\mathcal{Y}$ then do not match, communication ensues. We suggest three variants that minimize data movement, characterized by the number of synchronization barriers they require: zero, one, or $p-1$. Before describing these variants, we first motivate why it is sufficient to only consider one-dimensional partitionings of $\mathcal{A}$.

Assume a tensor of size $n = \prod_{k=1}^{d} n_k$, with $n_i \geq n_{i+1}$ for $i = 1, \ldots, d-1$, and $n_1 \geq p > 1$. Consider a series of d *TVMs*, $\mathcal{Y}_k = \mathcal{A} \times_k \mathbf{v}_k$, for all modes $k \in \{1, \ldots, d\}$. Assume *any* load-balanced distribution π of $\mathcal{A}$ and $\mathcal{Y}$ such that thread s has at most $2d\lceil n_1/p \rceil n/n_1$ work. For any $\mathbf{i} \in I$, the distribution π defines which thread multiplies the input tensor element $a_{\mathbf{i}}$ with its corresponding input vector element x_{i_k}. The thread(s) in $\pi(i_1, \ldots, i_{k-1}, I_k, i_{k+1}, \ldots, i_d)$ are said to *contribute* to the reduction of $y_{\mathbf{j}}$, where $\mathbf{j} = (i_1, \ldots, i_{k-1}, 1, i_{k+1}, \ldots, i_d)$, as they perform local reductions of multiplicands to the same element $y_{\mathbf{j}}$. We do not assume a specific reduction algorithm and count the minimal work involved.

For any $\mathbf{i} \in I$, let $J_{\mathbf{i}} = \{\mathbf{j} \in I \mid \vee_{k=1}^{d} i_k = j_k\}$ be the set of elements lying on d different axes which all go through $\mathbf{i}$, as illustrated in Figure 1 (left). Let $X_{\mathbf{i}} = \pi(J_{\mathbf{i}})$, where π is any distribution, describe the set of threads to which elements in $J_{\mathbf{i}}$ are mapped. Should $|X_{\mathbf{i}}| > 1$ for all $\mathbf{i} \in I$, then there is at least one *TVM* for which all elements of $\mathcal{Y}$ are involved in a reduction, as at least two threads contribute to $y_{\mathbf{j}}$. For a 1D distribution, this amounts to n/n_1 reductions, occurring only for mode 1, which shows that this lower bound on communication complexity for a series of *TVMs* is attainable. We will now consider if we can do better by allowing $\mathbf{i}$ for which $|X_{\mathbf{i}}| = 1$, and if so, by how much.

Suppose there exist $r = \prod_{k=1}^{d} r_k$ coordinates $\mathbf{i} \in I$ such that $X_{\mathbf{i}} = \{s\}$, which form a hyper-rectangular subtensor $\mathcal{B}$ of side length $r_k < n_k$ contained in $\mathcal{A}$, as in Figure 1 (right). We choose a hyper-rectangular shape, so that the r elements create the minimum amount of redundant work. Since $|X_{\mathbf{i}}| = 1$, the number of coordinates which must then also lie on thread s is $r(\sum_{k=1}^{d} n_k/r_k - (d-1))$. If $r_k = 2^{1/(d-1)} n_k/p^{1/(d-1)}$, this already corresponds to a load exceeding the assumed load balance $(2n - n/n_1)/p$; see Appendix A for details behind the constant factor $2^{1/(d-1)}$. Furthermore, with $r = 2^{d/(d-1)} n/p^{d/(d-1)}$ such coordinates, the lower bound on communication complexity may only be reduced to $n/n_1(1-2/p)$, where $r/r_1 = 2n/pn_1$ is the projection of the cube r onto the $d-1$-dimensional output tensor. The data movement on the input vector is at most $(d-1)n_1$, which typically is significantly less than the data movement associated with the output tensor. Thus, the $\pi_{1\mathrm{D}}$ distribution is asymptotically optimal when $n/n_1 \gg (d-1)n_1$ and $d > 2$.

3.3.1 0-sync.

We avoid performing a reduction on $\mathcal{Y}$ for $k = 1$ by storing $\mathcal{A}$ twice; once with a 1D distribution over mode 1, another time using a 1D distribution over mode d. Although the storage requirement is doubled, data movement remains minimal while explicit reduction for $k = 1$ is completely eliminated, since the copy with the 1D distribution over mode d can then be used without penalty. In either case, the parallel *TVM* computation completes after a sequential thread-local *TVM*; this variant requires no barriers to resolve data dependencies.

3.3.2 1-sync.

This variant performs an explicit reduction of the $\mathcal{Y}_s$ for $k = 1$ and behaves as the 0-sync variant otherwise. It requires a larger buffer for the $\mathcal{Y}_s$ to cope with $k = 1$, since each thread computes a full output tensor $\mathcal{A}_s \times_1 \mathbf{x}$ that contains partial results only. The output tensor is then reduced by all p threads, such that each thread contains its part according to a $\pi_{1\mathrm{D}}$ distribution over mode 2, i.e., $\mathcal{Y}_t = (\sum_{s=1}^{p}(\mathcal{A}_s \times_1 \mathbf{x}))_t$, for all $t \in P$. By load balance, each thread has at most $\lceil n_2/p \rceil n/(n_1 n_2)$ elements. A barrier must separate the local *TVM* from the reduction phase to ensure no incomplete $\mathcal{Y}_s$ are reduced.

3.3.3 q-sync.

This variant stores $\mathcal{A}$ with a 1D distribution over mode 1. It also stores two versions of the output tensor, one interleaved $\mathcal{Y}$ and one thread-local $\mathcal{Y}_s$. The vector $\mathbf{x}$ is interleaved. Both $\mathcal{A}_s$ and $\mathcal{Y}_s$ are split into $q = \prod_{i=2}^{d} q_i \geq p$ parts, by splitting each object into q_i parts across mode i. We index the resulting objects as $\mathcal{A}_{s,t}$, which are explicitly allocated to thread s, and $\mathcal{Y}_{s,t}$, which are both allocated as explicit and interleaved. The input vector $\mathbf{x}$ remains interleaved. The algorithm is seen below.

1: **if** $k = 1$ **then**
2: $\quad \mathcal{Y} = \mathcal{A}_{s,s} \times_k \mathbf{x}$
3: $\quad$ **for** $t = 2$ to q **do**
4: $\quad\quad$ **barrier**
5: $\quad\quad \mathcal{Y}$ += $\mathcal{A}_{s,(t+s-1) \bmod q+1} \times_k \mathbf{x}$
6: **else**
7: $\quad$ **for** $t = 1$ **to** q **do**
8: $\quad\quad \mathcal{Y}_{s,t}$ += $\mathcal{A}_{s,t} \times_k \mathbf{x}$

If this algorithm is to re-use output of mode-0 *TVM*, then, similarly to the 0-sync variant each thread must re-synchronize its local $\mathcal{Y}_{s,t}$ with $\mathcal{Y}$. Thus, unless the need explicitly arises, implementations need not distribute $\mathcal{Y}$ over n_2 as part of a mode-1 *TVM* (at the cost of interleaved data movement on $\mathcal{Y}$).

This algorithm avoids doubling the storage requirement yet still eliminates explicit reductions for $k = 1$, replacing reduction with synchronization. For $k > 1$ no barriers are required since each thread writes into disjoint areas of $\mathcal{Y}$ due to the 1D block distribution over mode 1. For simplicity, we omit the indexing of the interleaved $\mathcal{Y}$ into one of its q subtensors. Additionally, lines 7 and 8 assume the amount of subtensors in $\mathcal{Y}_s$ is the same as in $\mathcal{A}_s$, i.e., $q_k = 1$. Otherwise, the code has to be adapted to go over q/q_k output subtensors q_k times.

3.3.4 Interleaved $q(i)$-sync.

This q-sync variant assumes only an interleaved $\mathcal{Y}$ in lieu of thread-local $\mathcal{Y}_s$. Each $\mathcal{A}_s$ is further split into q parts, each stored thread-locally, giving rise to the input tensors $\mathcal{A}_{s,t}$; this split is equal across all threads and is used to synchronise writing into the output tensor. Each thread s executes:

1: $\mathcal{Y} = \mathcal{A}_{s,s} \times_k \mathbf{x}$
2: **for** $t = 2$ **to** q **do**

3: **if** $k = 1$ **then**
4: **barrier**
5: $\mathcal{Y} \mathrel{+}= \mathcal{A}_{s,(t+s-1) \bmod q+1} \times_k \mathbf{x}$

For simplicity, we omit any offset into $\mathcal{Y}$.

3.3.5 Explicit $q(e)$-sync.

The explicit q-sync variant allocates all $\mathcal{A}_s$ and $\mathcal{Y}_s$ explicitly local to thread s and keeps $\mathbf{x}$ interleaved. We split each $\mathcal{A}_s$ into q subtensors $\mathcal{A}_{s,t}$. Each thread s executes:

1: **if** $k = 1$ **then**
2: $\mathcal{Y}_{s,s} = A_{s,s} \times_k \mathbf{x}$
3: **for** $t = 2$ **to** q **do**
4: **barrier**
5: $\mathcal{Y}_{(t+s-1) \bmod q+1,t} \mathrel{+}= A_{s,t} \times_k \mathbf{x}$
6: **else**
7: **for** $t = 1$ **to** q **do**
8: $\mathcal{Y}_{s,t} = A_{s,t} \times_k \mathbf{x}$

An inter-socket data movement occurs on the output tensor at line 5. To cope with cases where the output tensors act as input on successive *TVM* calls on all possible modes, $\mathcal{A}_s$ must be split into $q \geq p$ parts *across each dimension*, while $\mathcal{Y}_s$ should likewise be split into at least p^{d-1} parts. We emphasize a careful implementation will never allocate p^d separate subtensors.

4 Analysis of the algorithms

We analyse the parallel *TVM* algorithms from the previous section, restricting ourselves not only to the amount of data moved during a *TVM* computation, but also consider mode-obliviousness, memory, and work. We divide data movement into intra-socket data movement (where cores contend for resources) and inter-socket data movement (where data is moved over a communication bus, instead of only to and from local memory). For quantifying data movement we assume perfect caching, meaning that all required data elements are touched exactly once. Once we quantify algorithm properties in each of these five dimensions, we consider their *iso-efficiencies* [3].

Consider the memory requirement M_s in number of words to store, and the number of barriers S. Each thread s thus executes $S+1$ different phases, which are numbered using integer i, $0 \leq i \leq S$. For each thread and phase, let intra-socket data movement $U_{s,i}$ and inter-socket data movement $V_{s,i}$ be in number of words, while work $W_{s,i}$ is in number of floating point operations (flops). These quantities fully quantify an algorithm, while the following quantifies a machine: intra-socket throughput g and the inter-socket throughput h, both in seconds per word (per socket); thread compute speed by r seconds per flop, and the time L in which a barrier completes in seconds.

Since barriers require active participation by processor cores while they also make use of communication, the time in which a given algorithm completes a *TVM* computation is given by

$$T(n,p) = \sum_{i=0}^{S} \left[\max_{k \in \{1,\ldots,p\}} (\max\{U_{k,i}g + V_{k,i}h, W_{k,i}r\}) + L \right]. \tag{1}$$

A perfect sequential *TVM* algorithm completes in $T_{\text{seq}}(n) = \max\{2nr, (n + n/n_k + n_k)g\}$ time. The *parallel overhead* $O(n,p)$ then is $pT(n,p) - T_{\text{seq}}(n)$ while the *parallel efficiency* $E(n,p)$ is the

speedup of an algorithm divided by p:

$$E(n,p) = T_{\text{seq}}(n)/pT(n,p) = 1/\left(\frac{O(n,p)}{T_{\text{seq}}(n)} + 1\right). \tag{2}$$

The above formula allows us to compute how fast n should grow to retain the same parallel efficiency as p increases and vice versa, giving rise to the concept of iso-efficiency. Strongly scaling algorithms have that $O(n,p)$ is independent of p (which is unrealistic), while weakly scalable algorithms have that the ratio $O(pn,p)/T_{\text{seq}}(pn)$ is constant; iso-efficiency instead tells us a much wider range of conditions under which the algorithm scales.

The *TVM* computation is a heavily bandwidth-bound operation. It performs $2n$ flops on $n + n/n_k + n_k$ data elements, and thus has arithmetic intensity equal to $1 < \frac{2n}{n+n/n_k+n_k} < 2$ flop per element. This amounts to a heavily bandwidth-bound computation even when considering a *sequential TVM* [9]. The multi-threaded case is even more challenging, as cores on a single socket compete for the same local memory bandwidth. In our subsequent analyses we will thus ignore the computational part of the equations for T, O, and E. We also consider memory overhead and efficiency versus the sequential memory requirement $M_{\text{seq}} = n + \max_k (n/n_k + n_k)$ words.

4.1 loopedBLAS

The *loopedBLAS* variant interleaves $\mathcal{A}$, $\mathcal{Y}$, and $\mathbf{x}$, storing them once while it performs $2n$ flops to complete the *TVM*; it is thus both memory- and work-optimal. It does not include any cache-oblivious nor mode-oblivious optimizations, and requires no barrier synchronisations ($S = 0$). Since all memory used is interleaved we assume their effective bandwidth is spread over g and h proportional to the number of CPU sockets p_{s}. Assuming a uniform work balance is achieved at run time, all p threads read n/p data from $\mathcal{A}$, write $n/(pn_k)$ to $\mathcal{Y}$, and read the full n_k elements of $\mathbf{x}$. Thus $U(s,0) = \frac{1}{p_{\text{s}}}v$ and $V(s,0) = \frac{p_{\text{s}}-1}{p_{\text{s}}}v$ with $v = \left(\frac{n+n/n_k}{p} + n_k\right)$ and the parallel overhead becomes

$$O(n,p) = \frac{p_{\text{s}}-1}{p_{\text{s}}}(n + n/n_k)(h-g) + n_k\left((p_t-1)g + p_t(p_s-1)h\right). \tag{3}$$

We see the overhead is dominated by $\mathcal{O}(n(h-g))$ as p_{s} increases, while for $p_{\text{s}} = 1$ the overhead simplifies to $\Theta(p_t n_k g)$. This excludes any underlying overhead of the parallel implementation of BLAS.

4.2 0-sync

This algorithm incurs n words of extra storage and thus is not memory optimal. In both cases of $k = 1$ and $k > 1$ the amount of work executed remains optimal at $2n$ flops. The cache- and mode-oblivious optimisations from our earlier work are fully exploited by this algorithm, while, like *loopedBLAS*, S remains zero. Here, $\mathcal{A}$ and $\mathcal{Y}$ are allocated explicitly while $\mathbf{x}$ remains interleaved; hence $U(s,0) = \frac{1}{p}(n + n/n_k) + \frac{1}{p_{\text{s}}}n_k$ and $V(s,0) = \frac{p_{\text{s}}-1}{p_{\text{s}}}n_k$, resulting in

$$O(n,p) = (p_{\text{t}}-1)n_k g + p_{\text{t}}(p_{\text{s}}-1)n_k h. \tag{4}$$

This overhead is bounded by $\Theta(pn_k h)$ for $p_{\text{s}} > 1$, a significant improvement over *loopedBLAS*.

If the output tensor $\mathcal{Y}$ must assume a similar datastructure to $\mathcal{A}$ for further processing, the output must also be stored twice. The minimal cost for this incurs extra overhead at $\Theta((n/n_k)g)$; i.e., a thread-local data copy, which remains asymptotically smaller than $T_{\text{seq}} = \mathcal{O}(ng)$. We do note that applications which require repeated *TVMs* such as the higher-order power method actually do *not* require this extra step; multilinearity can be exploited to perform the HOPM block-by-block [9, Section 5.6].

4.3 1-sync

This variant requires the $\mathcal{Y}_s$ are all of size n/n_k instead of $(n/n_k)/p$ elements, which constitutes a memory overhead of $(p-1)(n/n_k)$. It benefits from the same cache- and mode-oblivious properties as the 0-sync variant, and achieves the same overhead (Eq. 4) when $k > 1$. For $k = 1$, however, we must account for the reduction phase on the $\mathcal{Y}_s$ and for the barrier that precedes it. Reduction proceeds with minimal cost by having each thread reduce $(n/n_k)/p$ elements corresponding to those elements it should locally store, resulting in an overhead $O(n, p)$ for a mode-1 *TVM* of

$$p_t(n/n_1)g + p_t(p_s - 1)(n/n_1)h + (1 - 1/p_s)n_1(h - g) + (p - 1)(L + (n/n_1)r)$$

This extra overhead is proportional to $(n/n_1)/p$ for both memory movement and flops.

4.4 The interleaved $q(i)$-sync

The interleaved $q(i)$-sync variant requires only the interleaved storage of $\mathcal{Y}$, which implies writing output requires inter-process data movement for all modes, i.e., a significant and equal overhead for all modes. Only accesses to $\mathcal{A}_{s,p}$ remain explicit. This variant remains both memory and work optimal, while it incurs $q - 1$ barriers, and requires the complete output tensor to be accessed by all p threads, for $k = 1$. Thus, it results in an overhead $O(n, p)$ for a mode-1 *TVM* of

$$(p_t - 1)(n/n_1)g + p_t(p_s - 1)(n/n_1)h + (1 - 1/p_s)n_1(h - g) + p(p - 1)L,$$

which increases with $p(p - 1)L$. Hence, this parallel overhead is a significant increase over that of the 0-sync variant (Eq. 4) if $k = 1$, and equivalent otherwise. It still improves significantly over *loopedBLAS* (compare Eq. 3). For all other modes, the overhead is

$$O(n, p) = (1 - 1/p_s)(n/n_k + n_k)(h - g).$$

4.5 The explicit $q(e)$-sync and q-sync

In the explicit variant, and the q-sync variant, accesses to $\mathcal{A}_{s,p}$ and $\mathcal{Y}_{s,p}$ are explicit, while accesses to vector are interleaved. This variant remains work- and memory-optimal but reduces the parallel overhead for all other modes than 1 to

$$O(n, p) = (p_t - 1)n_k g + p_t(p_s - 1)n_k h.$$

This explicit variant improves on the interleaved one in that inter-socket communication related to $\mathcal{Y}$ is now only incurred for $k = 1$. Compared to the 0-sync variant, the q-sync variants trade synchronization and inter-socket communication for enhanced memory efficiency.

Method	Work	Memory	Movement	Barrier	Oblivious	Implicit	Explicit	k
loopedBLAS	**0**	**0**	$n(h-g)$	**0**	none	$\mathbf{x}, \mathcal{A}, \mathcal{Y}$	-	-
0-sync	**0**	n	$\mathbf{pn_1h + p_tn_1g}$	**0**	**full**	$\mathbf{x}$	$\mathcal{A}, \mathcal{A}, \mathcal{Y}$	-
1-sync	pn/n_1r	pn/n_1	$pn/n_1h + p_tn/n_1g$	pL	**full**	$\mathbf{x}$	$\mathcal{A}, \mathcal{Y}$	-
$q(i)$-sync	**0**	**0**	$pn/n_1h + p_tn/n_1g$	p^2L	good	$\mathbf{x}, \mathcal{Y}$	$\mathcal{A}$	1
$q(e)$-sync	**0**	**0**	$pn/n_1h + p_tn/n_1g$	p^2L	good	$\mathbf{x}$	$\mathcal{A}, \mathcal{Y}$	1
q-sync	**0**	n/n_1	$pn/n_1h + p_tn/n_1g$	p^2L	good	$\mathbf{x}, \mathcal{Y}$	$\mathcal{A}, \mathcal{Y}$	1

Table 2 – Summary of overheads for each parallel shared-memory *TVM* algorithm, plus the allocation mode of $\mathcal{A}, \mathcal{Y}$, and $\mathbf{x}$. We display the worst-case asymptotics; i.e., assuming $p_\mathrm{s} > 1$ and the worst-case k for non mode-oblivious algorithms. Optimal overheads are in bold.

4.6 Mode-obliviousness

The *loopedBLAS* algorithm is highly sensitive to the mode k in which a *TVM* is executed, while those algorithms based on the $\rho_Z\rho_\pi$ tensor layout are, by design, not sensitive to k [9]. The 0-sync and 1-sync variants exploit the $\rho_Z\rho_\pi$ maximally; the thread-local tensors use a single such layout, and each thread thus behaves fully mode-oblivious.

For the q-sync variants, however, each locally stored input tensor is split into subtensors. Suppose mode k has $\mathcal{A}_s$ split in q_k parts, resulting in $q = \prod_{i=2}^{d} q_i \geq p$ parts stored using a $\rho_Z\rho_\pi$ layout; depending on how these subtensors are ordered. This may hamper both cache efficiency and mode-obliviousness, since reading from $\mathbf{x}$ and writing to $\mathcal{Y}$ now only partially follows a Morton order. Hence, we should make sure to minimize q, and $\max_i q_i$– i.e., q-sync behaves optimally if the q_i are the result of an integer factorization of p.

4.7 Iso-efficiencies

Table 2 summarizes the results from this section. Note that the parallel efficiency depends solely on the ratio $O(n,p)$ versus $T_\mathrm{seq}(n)$; when considering the work- and communication-optimal 0-sync algorithm, efficiency thus is proportional to $\frac{pn_kh}{ng}$; i.e., 0-sync scales as long as p grows proportionally with n/n_k. For *loopedBLAS*, efficiency is proportional to $(h/g-1)(p_\mathrm{s}-1)/p_\mathrm{s}$. Since $h/g-1$ is constant, it drops as the number of sockets increases; *loopedBLAS* does not scale at all.

For 1-sync, iso-efficiency is attained whenever $\frac{p}{n_k}(r+h+g/p_s) + \frac{p}{n}L$ is constant; i.e., p should grow linearly with n when computation is latency-bound, and linearly with n_k otherwise. Both are unfavorable. All q-sync variants attain iso-efficiency when p grows linearly with n_k and p^2 grows linearly with n, also unfavorable.

Table 3 summarizes the overheads of each algorithms assuming $p_\mathrm{s} = 1$.

5 Experiments

We run our experiments on a number of different Intel Ivy Bridge nodes with different specifications summarized in Table 4. As we do not use hyperthreading, we limit the algorithms to use at most $p/2$ threads equal the number of cores (each core supports 2 hyperthreads). We measure the maximum bandwidth of the systems using several variants of the STREAM benchmark, reporting the maximum measured performance only.

Method	Work	Memory	Movement	Barrier	Oblivious	Implicit	Explicit	k
loopedBLAS	**0**	**0**	$\mathbf{pn_1g}$	**0**	none	$\mathbf{x}, \mathcal{A}, \mathcal{Y}$	-	-
0-sync	**0**	n	$\mathbf{pn_1g}$	**0**	**full**	$\mathbf{x}$	$\mathcal{A}, \mathcal{A}, \mathcal{Y}$	-
1-sync	pn/n_1r	pn/n_1	pn/n_1g	pL	**full**	$\mathbf{x}$	$\mathcal{A}, \mathcal{Y}$	-
$q(i)$-sync	**0**	**0**	pn/n_1g	p^2L	good	$\mathbf{x}, \mathcal{Y}$	$\mathcal{A}$	1
$q(e)$-sync	**0**	**0**	pn/n_1g	p^2L	good	$\mathbf{x}$	$\mathcal{A}, \mathcal{Y}$	1
q-sync	**0**	n/n_1	pn/n_1g	p^2L	good	$\mathbf{x}, \mathcal{Y}$	$\mathcal{A}, \mathcal{Y}$	1

Table 3 – Like Table 2, but assuming $p_{\mathrm{s}} = 1$.

						Bandwidth	
Node	CPU (clock speed)	p_s	p_t	p	Memory size (clock speed)	STREAM	Theoretical
1	E5-2690 v2 (3 GHz)	2	20	40	256 GB (1600 MHz)	76.7 GB/s	95.37 GB/s
2	E7-4890 v2 (2.8 GHz)	4	30	120	512 GB (1333 MHz)	133.6 GB/s	158.91 GB/s
3	E7-8890 v2 (2.8 GHz)	8	30	240	2048 GB (1333 MHz)	441.9 GB/s	635.62 GB/s

Table 4 – An overview of machine configurations used. Memory runs in quad channel mode on nodes 1,2, and 3, and in octa-channel on node 4. Each processor has 32 KB of L1 cache memory per core, 256 KB of L2 cache memory per core, and $1.25p_t$ MB of L3 cache memory shared amongst the cores.

The system uses CentOS 7 with Linux kernel 3.10.0 and software is compiled using GCC version 6.1. We use Intel MKL version 2018.2.199 for *loopedBLAS*. We also run with LIBXSMM version 1.9-864 for algorithms based on blocked layouts (0- and q-sync), and retain only the result with the library which runs faster of the two. To benchmark a kernel, we conduct 10 experiments for each combination of dimension, mode, and algorithm.

To illustrate and experiment with the various possible trade offs in parallel *TVM*, we implemented the baseline synchronization-optimal *loopedBLAS*, the work- and communication-optimal 0-sync variant, and the work-optimal q-sync variant. and investigate their performance and mode-obliviousness. We measure algorithmic performance using the formula for effective bandwidth (GB/s). We benchmark tensors of order-two up to order-5. We choose n such that the combined input and output memory areas during a single *TVM* call have a combined size of at least 10 GBs; The exact array of tensor sizes and block sizes are given in Table 5, and Table 6. respectively. The block sizes selected ensure that computing a *TVM* on such a block fits L3 cache. This combination of tensor and block sizes ensures all algorithms run with perfect load balance and without requiring any padding of blocks; to ensure this, we choose block sizes that correspond to 0.5–1 MB of our L3 cache; note that for our parallel *TVM* variants, the Morton order ensures the remainder cache remains obliviously well-used. We additionally kept the sizes of tensors equal through all pairs of (d, p_s), which enables comparison of different algorithms within the same d and p_s.

5.1 Single-socket results

Table 7 shows the experimental results for the single-socket of Node 1. Note that it drops to half of the peak numbers measured in Table 4. Note that as there is no inter-socket communication, all

d / Node	1	2	3
2	45600×45600 (15.49)	68400×68400 (34.86)	136800×136800 (139.43)
3	$1360 \times 1360 \times 1360$ (18.74)	$4080 \times 680 \times 4080$ (84.34)	$4080 \times 680 \times 4080$ (84.34)
4	$440 \times 110 \times 88 \times 440$ (13.96)	$1320 \times 110 \times 132 \times 720$ (102.81)	$1440 \times 110 \times 66 \times 1440$ (112.16)
5	$240 \times 60 \times 36 \times 24 \times 240$(22.25)	$720 \times 60 \times 36 \times 24 \times 360$(100.11)	$720 \times 50 \times 36 \times 20 \times 720$(139.05)

Table 5 – Table of tensor sizes $n_1 \times \cdots \times n_d$ per tensor-order d and the node as given in Table 4. The exact size in GBs is given in parentheses.

d / Node	1	2	3
2	570×570	570×570	570×570
3	$68 \times 68 \times 68$	$68 \times 68 \times 68$	$34 \times 68 \times 34$
4	$22 \times 22 \times 22 \times 22$	$22 \times 22 \times 22 \times 12$	$12 \times 22 \times 22 \times 12$
5	$12 \times 12 \times 12 \times 12 \times 12$	$12 \times 12 \times 12 \times 12 \times 6$	$6 \times 10 \times 12 \times 10 \times 6$

Table 6 – Table of block sizes $b_1 \times \cdots \times b_d$ per tensor-order d and the node as given in Table 4. Sizes are chosen such that all elements of a single block can be stored in L3 cache.

memory regions are exceptionally allocated locally for intra-socket experiments.

As the *loopedBLAS* algorithm relies on the unfold storage, whose structure does require a loop over subtensors for modes 1 and d; thus, no for-loop parallelisation is possible for these modes and the algorithm employs the internal MKL parallelization. Thus, the Table shows its performance is highly mode-dependent, and that the algorithms based on blocked $\rho_Z \rho_\pi$ storage perform faster than the *loopedBLAS* algorithm. The block Morton order storage transfers the mode-obliviousness to parallel *TVMs* (the standard deviation oscillates within 1%), as the Morton order induces mode-oblivious behavior on each core.

5.2 Inter-socket results

Table 8, 9, 10, and 11 show the parallel-*TVM* results on machines with different numbers of sockets for tensors of order-2, 3, 4, and 5, respectively. These runtime results show a lack of scalability of *loopedBLAS*. This is due to the data structures being interleaved (Section 4.1) instead of making use of a 1D distribution. Interleaving or not only matters for multi-socket results, but since Table 7

	Average performance			Sample stddev.		
d	*loopedBLAS*	0-sync	q-sync	*loopedBLAS*	0-sync	q-sync
2	40.23	42.28	**42.54**	0.63	**0.55**	0.65
3	36.43	39.34	**39.87**	24.93	2.55	**2.50**
4	37.63	39.02	**39.05**	21.29	**4.35**	4.40
5	34.56	36.53	**36.65**	22.43	5.14	**4.26**

Table 7 – Average effective bandwidth (in GB/s) and relative standard deviation (in %, versus the average bandwidth) over all possible $k \in \{1, \ldots, d\}$ of algorithms running on a single processor (Node 1). The highest bandwidth and lowest standard deviation for each d are stated in **bold**.

	Sample stddev.			Average performance		
algorithm / p_s	2	4	8	2	4	8
loopedBLAS	1.74	17.15	3.72	68.52	50.68	9.68
0-sync	**0.03**	**0.10**	**0.34**	**84.19**	**150.82**	**492.32**
q-sync	0.12	0.11	0.74	84.17	150.39	487.38

Table 8 – Average effective bandwidth (in GB/s) and relative standard deviation (in % of average) over all possible $k \in \{1, \ldots, d\}$ of order-2 tensors of algorithms executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3). The highest bandwidth and lowest standard deviation for different d are stated in **bold**.

	Sample stddev.			Average performance		
algorithm / p_s	2	4	8	2	4	8
loopedBLAS	9.57	16.52	23.05	63.89	55.68	13.66
0-sync	2.80	**1.38**	**3.42**	**77.06**	**145.07**	**467.31**
q-sync	**1.90**	3.86	6.56	76.27	143.17	441.65

Table 9 – Average effective bandwidth (in GB/s) and relative standard deviation (in % of average) over all possible $k \in \{1, \ldots, d\}$ of order-3 tensors of algorithms executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3). The highest bandwidth and lowest standard deviation for different d are stated in **bold**.

conclusively shows that approaches based on our $\rho_Z \rho_\pi$ storage remain superior on single sockets, we may conclude our approach is superior at all scales.

The performance drops slightly when d increases for all variants. This is inherent to the BLAS libraries handling matrices with a lower row-to-column ratio better than tall-skinny or short-wide matrices [9]—and this ratio increases when processing higher-order tensors.

For first-mode *TVMs*, the 0-sync algorithm slightly outperforms the q-sync, while they achieve almost identical performance for all the other modes. The cause lies with the 0-sync not requiring any synchronization for $k = 1$, and the effect is the 0-sync achieves the lowest standard deviation. Our measured performances are within the impressive range of 75–88%, 81–95%, and 66–77% of theoretical peak performance for node 1, 2, and 3, respectively.

Tables 12, 13, 14, and 15 display the parallel efficiency versus the performance of the q-sync on a single socket. Each node takes its own baseline since the tensor sizes differ between nodes as per Table 5; one can thus only compare parallel efficiencies over the *columns* of these tables, and cannot compare rows; we compare algorithms, and do not investigate inter-socket scalability.

The astute reader will note parallel efficiencies larger than one; these are commonly due to cache-effects, in this case likely output tensors that fit in the combined cache of eight CPUs, but did not fit in cache of a single CPU. These tests conclusively show that both 0- and q-sync algorithms scale significantly better than *loopedBLAS* for $p_s > 1$, resulting in up to 35x higher efficiencies (for order-4 tensors on node 3).

	Sample stddev.			Average performance		
algorithm / p_s	2	4	8	2	4	8
loopedBLAS	16.99	23.43	15.45	61.60	47.73	12.59
0-sync	**2.84**	**2.44**	**1.88**	**77.12**	**138.54**	**446.01**
q-sync	3.67	5.47	9.98	76.82	137.79	424.85

Table 10 – Average effective bandwidth (in GB/s) and relative standard deviation (in % of average) over all possible $k \in \{1, \ldots, d\}$ of order-4 tensors of algorithms executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3). The highest bandwidth and lowest standard deviation for different d are stated in **bold**.

	Sample stddev.			Average performance		
algorithm / p_s	2	4	8	2	4	8
loopedBLAS	15.37	19.70	32.03	56.11	54.04	12.43
0-sync	**3.47**	**5.01**	**5.02**	**71.71**	**129.80**	**421.98**
q-sync	4.17	9.37	14.83	71.65	129.60	397.25

Table 11 – Average effective bandwidth (in GB/s) and relative standard deviation (in % of average) over all possible $k \in \{1, \ldots, d\}$ of order-5 tensors of algorithms executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3). The highest bandwidth and lowest standard deviation for different d are stated in **bold**.

algorithm / p_s	2	4	8
loopedBLAS	0.81	0.31	0.02
0-sync	0.99	0.93	0.98
q-sync	0.99	0.93	0.97

Table 12 – Parallel efficiency of algorithms on order-2 tensors executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3), calculated against the single-socket runtime on a given node of q-sync algorithm on the same problem size and tensor order.

algorithm / p_s	2	4	8
loopedBLAS	0.80	0.34	0.03
0-sync	0.97	0.88	0.96
q-sync	0.96	0.87	0.91

Table 13 – Parallel efficiency of algorithms on order-3 tensors executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3), calculated against the single-socket runtime on a given node of q-sync algorithm on the same problem size and tensor order.

algorithm / p_s	2	4	8
loopedBLAS	0.79	0.28	0.03
0-sync	0.99	0.83	1.05
q-sync	0.98	0.82	1.00

Table 14 – Parallel efficiency of algorithms on order-4 tensors executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3), calculated against the single-socket runtime on a given node of q-sync algorithm on the same problem size and tensor order.

algorithm / p_s	2	4	8
loopedBLAS	0.77	0.32	0.05
0-sync	0.98	0.76	1.53
q-sync	0.98	0.76	1.44

Table 15 – Parallel efficiency of algorithms on order-5 tensors executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3), calculated against the single-socket runtime on a given node of q-sync algorithm on the same problem size and tensor order.

6 Conclusions

We investigate the tensor–vector multiplication operation on shared-memory multicore machines. Building on an earlier work, where we developed blocked and mode-oblivious layouts for tensors, we here explore the design space of parallel shared-memory algorithms based on this same mode-oblivious layout, and propose several candidate algorithms. After analyzing those for work, memory, intra- and inter-socket data movement, the number of barriers, and mode obliviousness, we choose to implement two of them. These algorithms, called 0-sync and q-sync, deliver close to peak performance on up four different systems, with 1, 2, 4, and 8 sockets, and surpass a baseline algorithm based on looped BLAS calls that we optimized.

For future work, we plan to investigate the use of the proposed algorithms in distributed memory systems. All proposed variants should work well, after modifying them to use explicit broadcasts of the input vector and/or explicit reductions on (parts of) the output tensor. While additional buffer spaces may be required, we expect that the other shared-memory cost analyses will naturally transfer to the distributed-memory case, and (thus) favor the 0-sync variant for speed and the q-sync variant when memory

Appendix A

Consider a d-dimensional hyperrectangular tensor $\mathcal{T}$ of size $n = \prod_{k=1}^{d} n_k$. Let there be a hyperrectangular subtensor $\mathcal{B}$ of size $r = \prod_{k=1}^{d} r_k$, contained in $\mathcal{T}$. The number of elements e in $\mathcal{T}$ that lie orthogonal to $\mathcal{B}$ in any of the d dimensions is $\sum_{k=1}^{d} rn_k/r_k - r(d-1)$, which simplifies to $r(\sum_{k=1}^{d} n_k/r_k - (d-1))$. Assuming $p \leq n_1$ (which is the largest dimension in $\mathcal{T}$), we know that

$\lceil n_1/p \rceil n/n_1 < 2n/p$:

$$\lceil n_1/p \rceil n/n_1 \leq (n_1 n + pn - n)/pn_1 \leq (2n - n/n_1)/p < 2n/p$$

We are looking for a minimal r for which

$$e = r(\sum_{k=1}^{d} n_k/r_k - (d-1)) > 2n/p$$

for $r, n, d, p \in \mathbb{N}$ and under the assumptions that $p > 1$, $n \geq p$, $d > 2$, and $r_k < n_k$.

A reasonable guess would be $r_k = cn_k/p^{1/(d-1)}$, where $p^{1/(d-1)} > c > 0$ is some to-be-determined constant (thus the total size $r = c^d n/p^{d/(d-1)}$, and $n_k/r_k = p^{1/(d-1)}/c$). Plugging this in:

$$\begin{aligned} c^d n/p^{d/(d-1)}(dp^{1/(d-1)}/c - (d-1)) &> 2n/p \\ c^{(d-1)} n(d - 2/c^{(d-1)})/p &> c^d n(d-1)/p^{d/(d-1)}, \end{aligned}$$

which we can further simplify to

$$c^{(d-1)} n(d - 2/c^{(d-1)})/p \geq c^d n(d-1)/p^{d/(d-1)}$$

which is satisfied when $c^{(d-1)}/p \geq c^d/p^{d/(d-1)}$ and $-2/c^{(d-1)} \geq -1$. From the second equation, $c^{d-1} \geq 2$ and thus $c \geq 2^{1/(d-1)}$. From the first equation, $p \geq c^{(d-1)}$ and thus $p \geq 2$ which is assumed. The total number of elements r we can take must be smaller than $2^{d/(d-1)} n/p^{d/(d-1)}$.

References

[1] B. W. Bader and T. G. Kolda. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM TOMS*, 32(4):635–653, December 2006.

[2] G. Ballard, N. Knight, and K. Rouse. Communication lower bounds for matricized tensor times Khatri-Rao product. In *2018 IPDPS*, pages 557–567. IEEE, 2018.

[3] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(3):12–21, 1993.

[4] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, Oct. 2017. ISSN 2475-1421.

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

[6] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *SC'15*, pages 76:1–76:12, 2015.

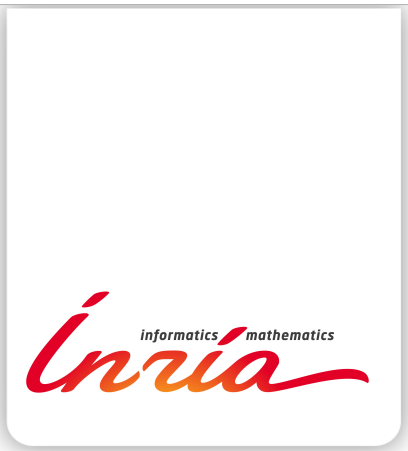
[7] D. Matthews. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing*, 40(1):C1–C24, 2018.

[8] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.

[9] F. Pawlowski, B. Uçar, and A. N. Yzelman. A multi-dimensional Morton-ordered block storage for mode-oblivious tensor computations. *Journal of Computational Science*, 2019. ISSN 1877-7503. doi: https://doi.org/10.1016/j.jocs.2019.02.007.

[10] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

[11] P. Springer and P. Bientinesi. Design of a high-performance gemm-like tensor–tensor multiplication. *ACM TOMS*, 44(3):28:1–28:29, 2018.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399