High performance tensor-vector multiplication on shared-memory systems

Filip Pawlowski, Bora Uçar, Albert-Jan Yzelman

High performance tensor–vector multiplication on shared-memory systems

Filip Pawłowski[1,2], Bora Uçar[2,3][0000−0002−4960−3545], and Albert-Jan Yzelman[1]

[1] Huawei Technologies France
20 Quai du Point du Jour, 92100 Boulogne-Billancourt, France
({filip.pawlowski1,albertjan.yzelman}@huawei.com)
[2] ENS Lyon (filip.pawlowski@ens-lyon.fr)
[3] CNRS and LIP (UMR5668, CNRS - ENS Lyon - UCB Lyon 1 - INRIA), Lyon, France (bora.ucar@ens-lyon.fr)

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.

Keywords: tensor computations· tensor–vector multiplication · shared-memory systems

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 [10]. 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. 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 [10]. The multi-threaded case is even more challenging, as cores on a single socket compete for the local memory bandwidth.

We proposed [10] a blocking approach for obtaining efficient, mode-oblivious tensor computations by investigating the case of tensor–vector multiplication in the single core setting. Earlier approaches to 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]. Recent alternatives [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, which we adopt for TVM. Other related work targets more complex operations [2] (called MTTKRP), and tensor–tensor multiplication [7, 11, 12]. Our TVM routines address a special case of TMM, which is a special case of TTM, based on our earlier work [10]. Apart from not explicitly considering TVM, these do not adapt the tensor layout. Kjolstad et al. [4] propose The Tensor Algebra Compiler (taco) for tensor computations, which generates straightforward for-loops in our case.

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 3.3. Section 4 contains experiments on up to 8-socket 120 core systems. A deeper analysis of algorithms and experiments appears in the accompanying technical report [9], which we refer to for the sake of brevity.

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 [9]). $\mathcal{A}$ 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 $\mathbf{j}$. 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 $\mathcal{A}$.

2.2 Sequential *TVM* and dense tensor memory layouts

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 [10], summarized below.

A layout ρ maps tensor elements onto an array of size $n = \Pi_{i=1}^{d} n_i$. Let $\rho_\pi(\mathcal{A})$ be a layout, and π an ordering (permutation) of $(1, \ldots, d)$ such that $\rho_\pi(\mathcal{A}) : (i_1, \ldots, i_d) \mapsto \sum_{k=1}^{d} \left((i_{\pi_k} - 1) \prod_{j=k+1}^{d} n_{\pi_j} \right) + 1$, 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]. 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 $\mathcal{A}_j \in \mathbb{R}^{b_1 \times \cdots \times b_d}$, where $j \in \{1, \ldots, \prod_{i=1}^{d} a_i\}$, 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 [10]. 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).

3 Shared-memory parallel *TVM* algorithms and analysis

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. The set of all possible thread IDs is $P = \{1, \ldots, p\}$. 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.

The pinning of the threads entails the notion of *explicit* versus *interleaved* memory use (see Section 3.1 of the accompanying technical report [9]).

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 (see Section 3.1 in the report [9]).

3.1 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.

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*.

3.2 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 [10]; 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.

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 $(2n - n/n_1)/p$. 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 $d > 2$.

In the following, we discuss two 1D algorithms, while our accompanying technical report [9, Section 3.3] contains three more.

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

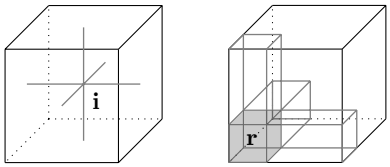

Fig. 1. Illustrations of elements in $J_{\mathbf{i}}$, indicated via thick gray lines, for an arbitrarily chosen $\mathbf{i}$ depicted by a filled dot (left), and for a cube of r elements $\mathbf{i}$ (right).

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.

***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 $\mathcal{Y}$).

3.3 Analysis

We investigate the amount of data moved during a *TVM* computation, 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. Since *TVM* is bandwidth bound, we consider memory overhead and efficiency versus the sequential memory requirement. Once we quantify algorithm properties in each of these five dimensions, we consider their *iso-efficiencies* [3]. Table 1 gives the summary, while the technical report contains an in-depth analysis [9, Section 4].

The *loopedBLAS* algorithm, thanks to interleaving, is both memory- and work-optimal. It does not include any cache-oblivious nor mode-oblivious optimizations, and has no barriers. Since all memory used is interleaved, the effective bandwidth is spread over intra and inter-socket bandwidth proportional to the

number of CPU sockets p_s. Thus, assuming a balanced work distribution, its overhead $\mathcal{O}((p_s-1)/p_s n(h-g))$ becomes $\Theta(n(h-g))$ as p_s increases. For $p_s = 1$ the overhead is $\Theta(p_t n_k g)$, which excludes any underlying overhead of its parallel implementation. The 0-sync algorithm is work optimal, incurs n words of extra storage (not memory optimal), and has no barriers. It fully exploits the cache- and mode-oblivious optimizations from our earlier work. The overhead of 0-sync is bounded by $\Theta(pn_k h)$ for $p_s > 1$, a significant improvement over *loopedBLAS*. The q-sync algorithm is work optimal, but not memory optimal as it stores $\mathcal{Y}$ twice. However, it improves upon 0-sync's overhead.

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**	$\mathbf{x}$	$\mathcal{A}, \mathcal{A}, \mathcal{Y}$	-
q-sync	**0**	n/n_d	$pn/n_dh + p_tn/n_dg$	p^2L	good	$\mathbf{x}, \mathcal{Y}$	$\mathcal{A}, \mathcal{Y}$	1

Table 1. Overheads of different *TVM* algorithms, and the allocation mode of $\mathcal{A}, \mathcal{Y}$, and $\mathbf{x}$. Optimal overheads are in **bold**. We display the worst-case asymptotics, i.e., assuming $p_s > 1$ and the worst-case k for non mode-oblivious algorithms. Intra-socket throughput g and the inter-socket throughput h are in seconds per word (per socket), threads' compute speed is in r seconds per flop, and a barrier completes in L seconds.

The *loopedBLAS* algorithm is highly sensitive to the mode k of the *TVM*, while the algorithms based on the $\rho_Z\rho_\pi$ tensor layout are not [10]. The 0-sync variant exploits the $\rho_Z\rho_\pi$ maximally; thus, it is fully mode-oblivious. In the q-sync variant, $\mathcal{A}_s$ are split into q parts, and each part is stored using a $\rho_Z\rho_\pi$ layout, which implies an overhead of $q-1$ space-filling curves. Furthermore, in the worst-case, each element of $\mathcal{Y}$ is touched $p-1$ times more than in a 0-sync variant, which hurts both cache efficiency and mode-obliviousness.

For *loopedBLAS*, efficiency is constant if $g/(h-g)$ decreases while p_s increases, which does not scale. The 0-sync attains efficiency when p grows linearly with n/n_k. The q-sync algorithm attains iso-efficiency when p grows linearly with n_k.

4 Experiments

We run our experiments on three Intel Ivy Bridge nodes, described in Table 2. In the paper, the terms KB, MB, and GB denote 2^{10}, 2^{20}, and 2^{30} bytes, respectively. We do not use hyperthreading and limit the tests to at most $p/2$ threads equal the number of cores (each core supports 2 hyperthreads). We measure the maximum bandwidth of the systems with the STREAM benchmark, and report the maximum measured performance. The system uses CentOS 7 with Linux kernel 3.10.0 and software is compiled with GCC version 6.1. We use Intel MKL version 2018.2.199 for *loopedBLAS*. For algorithms based on blocked layouts (0- and q-sync), we run with LIBXSMM version 1.9-864 and Intel MKL, and retain

the faster result. We conduct 10 experiments for each combination of dimension, mode, and algorithm and report the average performance (the effective bandwidth, GB/s) and its standard deviation among the modes.

We compare the synchronization-optimal *loopedBLAS*, the work- and communication-optimal 0-sync, and the work-optimal q-sync. We benchmark tensors of order 2 up to 5. We choose n such that the combined input and output memory areas during a single TVM call have a total size of at least 10 GBs. The exact tensor sizes and block sizes are given in Table 3, and Table 4, respectively. The block sizes selected ensure that computing a TVM on a block fits the L3 cache. This combination of tensor and block sizes ensures all algorithms run with perfect load balance and without requiring any padding of blocks. 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.

					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 2. Machine configurations used. Nodes 1 and 2 use a quad-channel and node 3 uses an octa-channel memory configuration. Each processor has 32 KB of L1 and 256 KB of L2 cache per core, and $1.25p_t$ MB of L3 cache shared amongst the cores.

d	Node 1	Node 2	Node 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 3. Tensor sizes $n_1 \times \cdots \times n_d$ per tensor-order d and node. The exact size in GBs is given in parentheses.

4.1 Single-socket results

Table 5 shows the results for the single-socket of node 1. Here, all memory regions are allocated locally. As *loopedBLAS* relies on the unfold storage and requires a loop over subtensors for modes 1 and d, no for-loop parallelization is possible for these modes, and MKL parallelization is used instead. Its performance is highly mode-dependent, and thus it is outperformed by the algorithms based on $\rho_Z\rho_\pi$-storage. The block Morton order storage transfers the mode-obliviousness to parallel $TVMs$ (the standard deviation oscillates within 1%).

d	Node 1	Node 2	Node 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 4. Block sizes $b_1 \times \cdots \times b_d$ per tensor-order d and node. Sizes are chosen such that all elements of a single block can be stored in L3 cache.

	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 5. Average effective bandwidth (in GB/s) and relative standard deviation (in % of the average) over all possible $k \in \{1, \ldots, d\}$ of algorithms on a single-socket of node 1. The highest bandwidth and lowest standard deviation for each d are in **bold**.

4.2 Inter-socket results

Table 6 shows results on the compute nodes for tensors of order-3 and 5 (the accompanying report [9] contains results for order-2 and 4 as well). These runtime results show a lack of scalability of *loopedBLAS*. This is due to the data structures being interleaved instead of making use of a 1D distribution. Interleaving or not only matters for multi-socket results, but since Table 5 conclusively shows that approaches based on our $\rho_Z \rho_\pi$-storage remain superior on single sockets, we conclude that our approach is superior at all scales. The performance drops slightly with the increasing d 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 [10].

	Order-3						Order-5					
	Sample stddev.			Avg. performance			Sample stddev.			Avg. performance		
algorithm / p_s	2	4	8	2	4	8	2	4	8	2	4	8
loopedBLAS	9.57	16.52	23.05	63.89	55.68	13.66	15.37	19.70	32.03	56.11	54.04	12.43
0-sync	2.80	**1.38**	**3.42**	**77.06**	**145.07**	**467.31**	**3.47**	**5.01**	**5.02**	**71.71**	**129.80**	**421.98**
q-sync	**1.90**	3.86	6.56	76.27	143.17	441.65	4.17	9.37	14.83	71.65	129.60	397.25

Table 6. Average effective bandwidth (in GB/s) and relative standard deviation (in % of average) over all possible $k \in \{1, \ldots, d\}$ of order-3 and -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**.

As 0-sync does not require any synchronization for $k = 1$, it achieves the lowest standard deviation. Thus, for *TVMs* of mode 1, the 0-sync algorithm slightly outperforms the q-sync, while they achieve almost identical performance for all the other modes. Some results are faster than STREAM benchmark, as the output tensor fits the combined L3 size and enables super-linear speedup. When passing from 4 to 8 processors the increase may be higher than twofold, due to the octa-channel memory on node 3. Overall, 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.

Table 7 displays the parallel efficiency on three different nodes 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 3; one can thus only compare parallel efficiencies over the *columns* of these tables. We compare algorithms, and do not investigate inter-socket scalability. The efficiencies larger than one are commonly due to cache-effects; here, output tensors 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 looped-BLAS for $p_s > 1$, resulting in up to 35x higher efficiencies (for order-4 tensors on node 3).

	Order-2			Order-3			Order-4			Order-5		
algorithm / p_s	2	4	8	2	4	8	2	4	8	2	4	8
loopedBLAS	0.81	0.31	0.02	0.80	0.34	0.03	0.79	0.28	0.03	0.77	0.32	0.05
0-sync	0.99	0.93	0.98	0.97	0.88	0.96	0.99	0.83	1.05	0.98	0.76	1.53
q-sync	0.99	0.93	0.97	0.96	0.87	0.91	0.98	0.82	1.00	0.98	0.76	1.44

Table 7. Parallel efficiency of algorithms on order-2 up to 5 tensors executed on 3 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.

5 Conclusions

We investigate the tensor–vector multiplication operation on shared-memory systems. Building on an earlier work, where we developed blocked and mode-oblivious layouts for tensors, we explore the design space of parallel shared-memory algorithms based on this same mode-oblivious layout, and propose several parallel 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 three different systems, using 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.

References

1. Bader, B.W., Kolda, T.G.: Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. ACM TOMS **32**(4), 635–653 (2006)
2. Ballard, G., Knight, N., Rouse, K.: Communication lower bounds for matricized tensor times Khatri-Rao product. In: 2018 IPDPS, pp. 557–567. IEEE (2018)
3. Grama, A.Y., Gupta, A., Kumar, V.: Isoefficiency: Measuring the scalability of parallel algorithms and architectures. IEEE Parallel & Distributed Technology: Systems & Applications **1**(3), 12–21 (1993)
4. Kjolstad, F., Kamil, S., Chou, S., Lugato, D., Amarasinghe, S.: The Tensor Algebra Compiler. Proc. ACM Program. Lang. **1**(OOPSLA), 77:1–77:29 (2017)
5. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM Review **51**(3), 455–500 (2009)
6. Li, J., Battaglino, C., Perros, I., Sun, J., Vuduc, R.: An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In: SC'15, pp. 76:1–76:12 (2015)
7. Matthews, D.: High-performance tensor contraction without transposition. SIAM Journal on Scientific Computing **40**(1), C1–C24 (2018)
8. Morton, G.M.: A computer oriented geodetic data base and a new technique in file sequencing (1966)
9. Pawłowski, F., Uçar, B., Yzelman, A.J.N.: High performance tensor–vector multiples on shared memory systems. Tech. Rep. 9274, Inria, Grenoble-Rhône-Alpes (2019)
10. Pawlowski, F., Uçar, B., Yzelman, A.N.: A multi-dimensional Morton-ordered block storage for mode-oblivious tensor computations. Journal of Computational Science (2019). https://doi.org/https://doi.org/10.1016/j.jocs.2019.02.007
11. Solomonik, E., Matthews, D., Hammond, J.R., Stanton, J.F., Demmel, J.: A massively parallel tensor contraction framework for coupled-cluster computations. Journal of Parallel and Distributed Computing **74**(12), 3176–3190 (2014)
12. Springer, P., Bientinesi, P.: Design of a high-performance gemm-like tensor–tensor multiplication. ACM TOMS **44**(3), 28:1–28:29 (2018)