A native tensor–vector multiplication algorithm for high performance computing

Pedro J. Martinez-Ferrer, A. N. Yzelman and Vicenç Beltran

(Accepted article, published journal article available at https://doi.org/10.1109/TPDS.2022.3153113)

Abstract—Tensor computations are important mathematical operations for applications that rely on multidimensional data. The tensor–vector multiplication (TVM) is the most memory-bound tensor contraction in this class of operations. This paper proposes an open-source TVM algorithm which is much simpler and efficient than previous approaches, making it suitable for integration in the most popular BLAS libraries available today. Our algorithm has been written from scratch and features unit-stride memory accesses, cache awareness, mode obliviousness, full vectorization and multi-threading as well as NUMA awareness for non-hierarchically stored dense tensors. Numerical experiments are carried out on tensors up to order 10 and various compilers and hardware architectures equipped with traditional DDR and high bandwidth memory (HBM). For large tensors the average performance of the TVM ranges between 62% and 76% of the theoretical bandwidth for NUMA systems with DDR memory and remains independent of the contraction mode. On NUMA systems with HBM the TVM exhibits some mode dependency but manages to reach performance figures close to peak values. Finally, the higher-order power method is benchmarked with the proposed TVM kernel and delivers on average between 58% and 69% of the theoretical bandwidth for large tensors.

Index Terms—Parallel algorithms, shared memory, tensor computations, high bandwidth memory, NUMA

1 Introduction

The numerical application of multilinear algebra is present in a wide variety of scientific domains such as data mining and analysis [1], [2]. It is based on the notion of tensors (or multidimensional arrays), their properties and the operations defined on them. The tensor contraction is a fundamental operation since it is ubiquitous in many algorithms [3]. Examples of tensor contraction operations include the tensor–tensor multiplication (TTM), the tensor–matrix multiplication (TMM) and the tensor–vector multiplication (TVM). The TTM and TMM algorithms can be generalized to BLAS level 3 functions while the TVM can be classified as a BLAS level 2 kernel.

This paper focuses on the TVM algorithm applied to dense tensors as well as its implementation within the higher-order power method (HOPM) algorithm [4], which relies upon a series of TVM operations. The TVM involves a contraction over a unique mode of the tensor, hence turning out to be the most memory-bound numerical kernel among the three core tensor operations mentioned above. The TTM/TMM is analogous to the matrix–matrix multiplication (MMM) and the TVM to the matrix–vector multiplication (MVM). The MMM and MVM are widely known by the function names `gemm` and `gemv`, respectively, where the former is compute-bound and the latter is memory-bound. For a TVM contraction over mode k the arithmetic intensity (AI) is $2n/(n + n/n_k + n_k)$ FLOPs per item being n and n_k the total number of elements of the involved tensor and vector, respectively. This results in an AI bounded between 1 and 2 corresponding to the most extreme cases $n \gg n_k$ and $n = n_k^2$, respectively. For this reason, the throughput of a TVM execution is often measured in terms of bandwidth (e.g. GB/s). Another aspect worth considering in tensor computations, although it also applies to matrix computations to a much lesser extent, is mode obliviousness. This property guarantees that a tensor operation, e.g. a TVM, yields roughly similar performance independently of the contraction mode it is applied to.

It can be readily seen that both performance and mode obliviousness are crucial for efficient tensor algorithms. In the case of the TVM, previous studies focused on sequential execution have demonstrated that even the most widely extended "loop" and "unfold" TVM implementations are mode-dependent and deliver poor performance overall [5]. This is due to how tensors are stored in system memory. While matrices are commonly stored following a row-major or column-major ordering, a tensor of rank d can be stored as a multidimensional array using $d!$ different combinations although, in practice, only the first-order or last-order layouts are considered.

Another important observation with regard to the TVM efficiency is that, regardless of the tensor ordering, researchers do express the TVM algorithm as a series of calls to `gemm` [6] or `gemv` [7], [8] kernels over contiguous matrices

Accepted article by IEEE Transactions on Parallel and Distributed Systems 17 February 2022.

This work was supported in part by MCIN/AEI and ESF under Grant RYC2019-027592-I, and in part by the HPC Technology Innovation Lab, a Barcelona Supercomputing Center and Huawei research cooperation agreement (2020). (Corresponding author: Pedro J. Martinez-Ferrer.)

Pedro J. Martinez-Ferrer is with the Barcelona Supercomputing Center (BSC), Barcelona, Spain and also with the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain (e-mail: pedro.martinez.ferrer@bsc.es).

Albert-Jan Yzelman is with the Computing Systems Laboratory, Huawei Technologies Switzerland AG, Zürich (email: albertjan.yzelman@huawei.com).

Vicenç Beltran is with the Barcelona Supercomputing Center (BSC), Barcelona, Spain (e-mail: vicenc.beltran@bsc.es).

that represent the tensor. In multi-threaded scenarios this approach is prone to load imbalance as it may not be possible to evenly distribute these matrices among all the available threads [9]. This constitutes a potential source of performance loss. The present study contributes to enhance the TVM performance by:

- Designing from scratch an open-source algorithm exclusive to the TVM that is different from previous implementations based on external BLAS routines.
- Applying specific optimizations to this algorithm to enable full vectorization, mode obliviousness, multi-threading and NUMA awareness.
- Performing exhaustive benchmarks on several architectures using different compilers to compare its parallel performance against the theoretical bandwidth values reported by the manufacturers.

The remainder of this paper is organized as follows. Section 2 and 3 present the background and related work, respectively. The proposed TVM algorithm is introduced in Section 4 together with its integration into the HOPM benchmark. Section 5 evaluates in detail the performance of the TVM and HOPM algorithms and Section 6 presents the main conclusions and future work.

2 Background

This section has been made intentionally concise and the reader is referred to Kolda & Bader [3] for a more detailed description of tensors and their operations. In this work scalars, vectors, matrices and tensors follow the notation x, $\mathbf{x}$, $\mathbf{X}$ and $\mathcal{X}$, respectively. The d-order dense tensors considered herein can be interpreted as d-dimensional arrays which are stored in system memory according to the last-order storage format layout given by the tuple $\pi_d = (d-1, d-2, \ldots, 0)$. This decision is determined by the C++ programming language: we adopt the row-major order for storing multi-dimensional arrays and start counting or indexing elements from zero.

Consider a tensor $\mathcal{A} \in \mathbb{R}^{n_0 \times n_1 \times \cdots \times n_{d-1}}$ composed of d modes, that is a d-order tensor, and a vector $\mathbf{x} \in \mathbb{R}^{n_k}$ of n_k elements with $0 \leqslant k \leqslant d-1$. The k-mode tensor–vector multiplication is defined as $\mathcal{Y} = \mathcal{A} \times_k \mathbf{x}$ and results in another tensor $\mathcal{Y} = \mathbb{R}^{n_0 \times \cdots \times n_{k-1} \times 1 \times n_{k+1} \times \cdots \times n_{d-1}}$ composed of $d-1$ modes and whose kth mode is of size unity. Let $n = \prod_{i=0}^{d-1} n_i$, $l = n/n_k$ and $\mathbf{A}^{l \times n_k}$ (and its transpose $\mathbf{A}^{n_k \times l}$) be the matricization of $\mathcal{A}$ which reinterprets that tensor as an $l \times n_k$ matrix (and $n_k \times l$ matrix). A particularity of the matricized form of a tensor is that its elements do not need to be ordered consecutively (see for instance Fig. 1). A TVM algorithm can be thought as one or more instances of a matrix–vector multiplication on contiguous regions of the matricized form of a tensor. Since tensors are stored following a last-order layout, left-hand sided vector–matrix multiplications (VMMs, $\mathcal{Y} = \mathbf{x}^{\intercal} \times \mathbf{A}^{n_k \times l}$) are necessary for modes $k < d-1$ while a single matrix–vector multiplication (MVM, $\mathcal{Y} = \mathbf{A}^{l \times n_k} \times \mathbf{x}$) is required for the last contraction mode $k = d-1$.

We refer to `getvm_loop` and `getvm_unfold` to designate two common algorithms used to compute the TVM; see Pawłowski et al. [5] for details. The main difference between the two is that the former applies a VMM on consecutive portions of the matricized tensor $\mathbf{A}^{n_k \times l}$ while the latter reorders (i.e. unfolds) the tensor in memory to guarantee that all data is contiguous in system memory before carrying out a single VMM over the entire matrix. However, such unfolding yields an overhead over the looped algorithm and hence we discard it in this study. From a high performance computing perspective, a parallel implementation of the loop TVM imposes two nested levels: (i) the outer level corresponds to a loop over the aforementioned consecutive parts of the matricized tensor and (ii) the inner level which corresponds to the VMM algorithm itself. In order to exploit nested parallelism, researchers employ OpenMP to parallelize the outer loop and rely on parallel BLAS level 2 routines available in heavily optimized libraries [9]. Nested parallelism helps to increase the overall performance of the TVM loop algorithm but it also has caveats. For instance, one must depend on two different parallel strategies due to nesting making the algorithm prone to load balance issues.

3 Related work

The literature is populated with examples of TVM implementations that simply consist of interfaces to either BLAS level 3 [10] or level 2 kernels [5], [11]. Such implementations largely benefit from libraries like Intel MKL [12], LIBXSMM [13] and also BLIS [14] which offer heavily optimized BLAS kernels that are crucial for high performance. As explained in the previous section, it is fairly common for some TVM implementations to resort to tensor unfolding operations with their subsequent overheads while others employ a looped algorithm to operate on parts of the tensor so that no unfolding is necessary. All things considered, the aforementioned TVM algorithms have been mainly motivated by performance gains and cache-oblivious behaviour. Pawłowski et al. [5] were the first authors to assess mode obliviousness for the TVM algorithm by storing tensors following a Morton-order layout.

On the other hand, there has been extensive work put into libraries that deal natively with tensors. For example, TACO is a library for performing tensor algebra computations that automatically generates efficient code [15]. However, the TVM contraction defined therein sometimes can lack performance by not reverting to optimized BLAS level 2 kernels within its generated codes [5]. More recent libraries such as TBLIS [16] applies the BLIS philosophy to generate efficient tensor algebraic reductions thus eliminating the possible storage and performance overheads of resorting to external BLAS routines. Similarly, the authors of the framework GETT [6] propose a tensor-contraction generator to systematically reduce a tensor contraction to loops around a highly tuned matrix–matrix multiplication kernel (`gemm`) that delivers great single-threaded performance for single and double precision computations. Motivated by the fact that level-3 tensor contractions based on `gemm` do not perform equally well for tensor–vector multiplications, Bassoy [11] suggests a series of contraction algorithms that either directly call `gemv` kernels or recursively apply `gemv` on tensor slices multiple times. It is worth noting that such algorithms depend on an external BLAS level 2 routine provided by the OpenBLAS library.

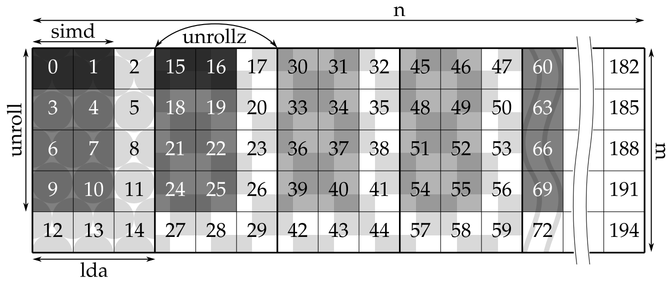


Fig. 1: Memory addresses following a last-order layout for the matricized form $\mathbf{A}^{5\times39}$ derived from the 1-mode TVM applied to the third-order tensor $\mathcal{A}^{13\times5\times3}$. The execution pattern of `getvm` shown here arises from parallelizing the entire TVM operation over columns (`n`) using three threads.

To the best of our knowledge there are no bibliographic works proposing a TVM algorithm independent of third-party BLAS libraries featuring mode obliviousness, multi-threading and NUMA awareness for non-hierarchically ordered dense tensors. In this regard the present study not only seeks to compare the TVM performance against the theoretical bandwidth of different hardware architectures, it can also be seen as a software tool to be used by compiler designers to optimize their generated binaries.

4 Algorithm Description

We proceed by describing our algorithm on a single NUMA node, extending it to multiple NUMA nodes and, finally, integrating it into the HOPM benchmark.

4.1 Single- and multi-threaded implementations

Contrary to previous approaches, a native algorithm for the k-mode TVM expresses the operation on *subsets* of both the input and output tensors. This is analogous to applying the BLAS level-2 routine `gemv` on portions of the input matrix and output vector. Based on this premise we propose a TVM native function `getvm` whose definition adopts the following schematic form in C++ syntax:

```
template<class objT> void getvm(
    intT layout, intT trans, intT m   , intT n,
    objT alpha , objT * a   , intT lda ,
               , objT * x   , intT incx,
                 objT * y0 ,
    objT beta  , objT * y   , intT incy);
```

Such a function shares the same list of arguments as `gemv`, plus an additional term `y0` which is a pointer to the very first element of the output tensor $\mathcal{Y}$. The variable `a` refers to the input tensor $\mathcal{A}$ and `x` corresponds to the input vector $\mathbf{x}$. The primitive data type `intT` represents an integer of any size while `objT` can be either an integer or a floating-point number of any given precision (for the sake of briefness, only floating-point tensors are considered in this work). This native definition brings us the opportunity to build our own VMM and MVM kernels from scratch, i.e. `vecMat` and `matVec`, in order to exploit novel optimization techniques for further single-threaded performance.

To illustrate the proposed algorithm, consider the following example of a 1-mode TVM on a third-order dense tensor $\mathcal{A}^{13\times5\times3}$ of doubles or 8-byte integers. Its matricized form has 5 rows and 39 columns whose layout in system memory is partially represented in Fig. 1. Recall that the tensor is stored in a generalized row-major format and thus Fig. 1 shows a logical view of $\mathcal{A}$ under a 1-mode TVM. Assume that the TVM is only to be applied to a subset of $\mathbf{A}$; for instance, it is executed over the first 13 columns of that matrix. Figure 1 represents indeed that subset composed of 5×13 elements. The corresponding portion of $\mathcal{Y}$, not shown here for the sake of conciseness, contains the first 13 elements of that tensor. Regardless of the size of the subset, each one of them is composed of up to three clearly differentiated zones: (i) a left zone marked in Fig. 1 with a pattern filled with packed circles, (ii) a certain number of central zones represented with a checkerboard pattern and (iii) a right zone marked with a wavy pattern.

Pointers `y` and `y0` are employed to index the zones of the tensor, which permits our single-threaded algorithm `getvm` to traverse all of them, one by one and from left to right, performing a vector–matrix multiplication in this case. While doing so, our `vecMat` routine exploits vectorization with the aid of OpenMP directives as well as unroll & jam techniques. The black (e.g. 128-bit SIMD/SVE) and gray (e.g. `unroll=4`) boxes represent the memory touched by these two techniques on each zone during the first *vector* iteration. An important optimization for the TVM native algorithm arises from adding a *second* unroll to the central zones (e.g. `unrollz=3`, light gray boxes). Should all the elements touched along this third dimension remain within L1 CPU cache, this technique economizes function calls which directly translates into greater performance, especially for higher-order tensors and large contraction modes. In this regard, the proposed function `vecMatz` can offer a competitive advantage over the conventional `gemv` routine for all internal modes $0 < k < d-1$.

For the first mode a single VMM is carried out via `vecMat` and for the last mode a single MVM is executed via `matVec`. The matrix–vector multiplication algorithm shares similar vectorization and unroll & jam optimizations. Furthermore, both algorithms benefit from C++ template arguments to generate, at compile time, *versioned* functions which deal with remainder loops issued from unrolling (as seen in Fig. 1) thus mitigating any performance degradation on these particular cases.

It is worth mentioning that our TVM library already provides optimized `getvm` versions for both aligned and unaligned memory accesses and the two most common usages, that is $\alpha = 1$ and $\beta = 0$ or $\beta = 1$, while it also supports the general expression $\mathcal{Y} := \alpha\mathcal{A} \times_k \mathbf{x} + \beta\mathcal{Y}$ which can access the involved arrays with non-unit strides (`incx`, `incy`). At this moment, the library only supports dense tensors stored with last- or row-major layout (`layout=0`) but extending it to first- or column-major ordering should be straightforward. Lastly, changing the floating-point precision can be simply done by redefining the primitive object `floatT` (e.g. via a preprocessor directive). The number of elements that fit within the vector length is automatically adjusted to guarantee fully vectorized code via OpenMP directives and the same applies to the memory alignment. This open-source TVM library (Version 1.0; Martinez-Ferrer [17]) is released under the GPLv3 license.

The multi-threaded implementation of `getvm` is pretty straightforward. Using the previous example as a reference, Code 1: Multi-threaded implementation of the native algorithm `getvm` by means of OpenMP "parallel for" pragma directives for the matrix–vector and vector–matrix multiplication variants.

```
 1  assert(layout == iZero);  // Ensure row-major layout: last-order
 2  if    (trans  == iZero) { // Matrix-vector (non-transposed vector)
 3    const intT bsM = splitInterval(m, nbA, unroll);
 4
 5    #pragma omp parallel for firstprivate(x, a, y)
 6    for(intT i = iZero; i < m; i += bsM) {
 7      const intT bsMA = MIN(bsM, m - i);
 8      getvm(layout  , trans, bsMA, n  ,
 9               alpha, a + i*lda, lda  , x, incx,
10           y, beta , y + i    , incy);
11    }
12  }
13  else if(trans == iOne) {  // Vector-matrix (transposed vector)
14    const intT bsN   = splitInterval2(n, nbA, simdVMsize/objTsize),
15               zsize = m*lda;
16
17    #pragma omp parallel for firstprivate(x, a, y)
18    for(intT j = iZero; j < n; j += bsN) {
19      const intT bsNA = MIN(bsN, n - j);
20      getvm(layout  , trans, m, bsNA,
21               alpha, a + zsize*(j/lda) + j % lda, lda  , x, incx,
22           y, beta , y + j                     , incy);
23    }
24  }
```

the matricized form of the input tensor $\mathbf{A}^{5\times 39}$ can be viewed as a collection of three subsets, each one composed of 13 columns. One can assign a unique thread to each portion of $\mathbf{A}$ in order to carry out the three TVM executions simultaneously since they constitute an embarrassingly parallel operation. Code 1 shows the multi-threaded implementation of the `getvm` algorithm via the OpenMP "parallel for" clause for both the matrix–vector (`trans=0`) and vector–matrix (`trans=1`) multiplication variants. The MVM case is covered in lines 8–10 and is similar to a `gemv` kernel with the extra argument `y` at the beginning of line 10. For the VMM operation one needs to set the appropriate offset to the input tensor `a` (line 21). The functions `splitInterval` attempt to divide the total number of rows (`m`) or columns (`n`) by the number of available threads (`nbA`) so that the result is a multiple of the unroll factor or the vector length, respectively, in order to maximize the algorithm performance. Critically, the multi-threaded implementation of the native algorithm presents a *flat* loop structure in contrast to the TVM loop algorithm which relies on 2-level nested parallelism [5]. This ultimately renders `getvm` easier to implement, easier to execute in parallel and more robust vis-à-vis load imbalance.

4.2 NUMA awareness support for TVM

The algorithm described in Code 1 is executed in parallel in a shared memory system via a single OpenMP parallel for loop. When dealing with NUMA architectures one can expect performance penalties if inter-NUMA communication takes place during the TVM operation [9]. The amount of communication is closely related to (i) the contraction mode of the TVM and (ii) how the system allocates the arrays `a` and `y` given by their first-touch policy, taken into account that memory is allocated on the node beloging to the thread that first accesses a memory page [18]. The first contraction mode is especially critical because, if the array `a` is uniformily allocated across all NUMA nodes,

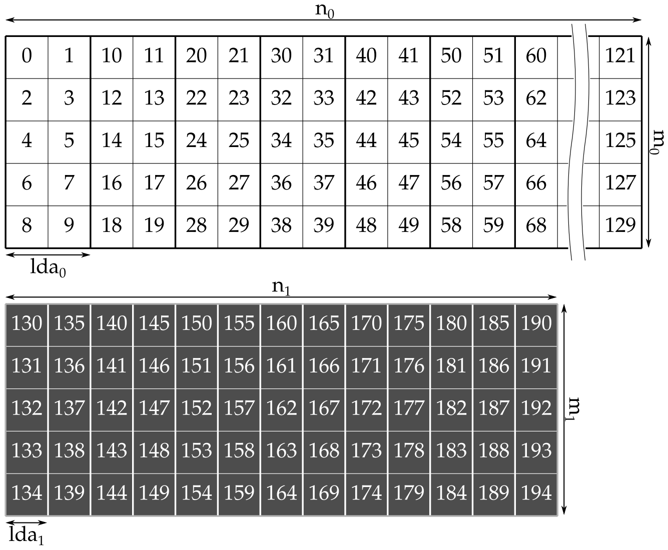


Fig. 2: Memory addresses (last-order layout) for the matricized form of a third-order dense tensor $\mathcal{A}^{13\times 5\times 3}$ disjointed along its last mode. Subtensors $\mathcal{A}_0^{13\times 5\times 2}$ (top) and $\mathcal{A}_1^{13\times 5\times 1}$ (down) are allocated on NUMA nodes 0 and 1, respectively. This particular matricized form corresponds to the 1-mode TVMs $\bigcup_{i=0}^{1} \mathcal{Y}_i = \bigcup_{i=0}^{1} \mathcal{A}_i \times_1 \mathbf{x}$ executed by `getvmd`.

applying `getvm` on different partitions of `a` would yield a significant amount of inter-NUMA communication and therefore the algorithm will not scale [9]. For the remaining contraction modes, communication reduces so dramatically that the impact on performance is negligible. In this study we propose two different strategies to render our TVM algorithm NUMA-aware.

The first method is the most straightforward, it remains transparent or implicit to the user and also yields better scalability results. By means of the function `mmap` we align all buffers to the system page size which is typically 2 MiB if transparent huge pages (THP) are active. Next, we carry out a first-touch policy for the actual allocation of buffers which modestly consists of initializing them to zero by following the exact same memory access pattern involved in the actual TVM operation (see Fig. 1). This first touch is done in parallel using the same hardware resources that are exploited later on by the k-mode TVM thus removing intra-NUMA communication on `a` for a given value of k. This method does not modify the original memory layout of the tensor and is supported by the multi-threaded implementation of `getvm` shown in Code 1.

The second approach explicitly splits or disjoints the original tensor across different NUMA nodes. This modifies the original memory layout in order to carry out the computation in parallel without NUMA interference and is partially based on earlier work [9]. Its implementation in a global address space is far more complex than the first approach. What is more, we only consider one-dimensional splitting: that is, tensors are disjointed along *one* of their modes. Depending on the contraction mode, tensors are disjointed on their last mode for $k < d-1$ and on their first mode for $k = d-1$. To better illustrate this, consider disjointing the input tensor from the previous 1-mode TVM

(Section 4.1) across two NUMA nodes. Such splitting yields two subtensors, i.e. $\mathcal{A}^{13\times5\times3} = \mathcal{A}_0^{13\times5\times2} \bigcup \mathcal{A}_1^{13\times5\times1}$, whose matricized form in system memory is represented in Fig. 2. To achieve this, the first 130 elements of buffer `a` need to be first-touched by threads belonging to the first NUMA node while the remaining elements are touched by threads of the other node. The same decomposition is carried out on buffer `y`. Note also that there is only one single buffer per tensor independently of the number of NUMA partitions. Translating between such an SPMD-like disjointed view and a standard last-order layout is by no means trivial. For example, the third column of $\mathbf{A}$ in Fig. 1 (addresses 2, 5, 8, 11, 14) corresponds to the first column of $\mathbf{A}_1$ located at node 1 in Fig. 2 (addresses 130, 131, 132, 133, 134). Furthermore, the amount of load imbalance this method can incur in our experience often results in subpar performance with respect to the first NUMA-aware strategy presented. Lastly, we note that both strategies presented achieve the asymptotically optimal data movement bound for a sequence of $(0, 1, \ldots, d-1)$-mode TVMs derived by Pawłowksi et al. [9].

The multi-threaded function `getvmd` adds support for disjointed tensors. It accepts the same arguments as `getvm` although variables `m`, `n` and `lda` are no longer primitive objects; instead, they are pointers to lists of integers with a size equal to the number of NUMA partitions. This multi-threaded function relies on two-level nested parallelism in which the first level corresponds to an outer loop along NUMA nodes. Here we rely on OpenMP to specify the correct affinity in order to ensure that each iteration of the loop is assigned to a distinct node. At the same time, each iteration of this outer loop contains a call to the regular `getvm` function which itself utilizes all the hardware resources available on that particular NUMA partition.

This section concludes with a brief mention to the role of the input vector $\mathbf{x}$ in achieving NUMA awareness. In practice, this variable is orders of magnitude smaller than the tensors involved in the TVM operation and, for this reason, its influence on performance is negligible. In this work, the array `x` is uniformily distributed among all nodes and thus it is subject to inter-NUMA communication. Nevertheless, the communication of buffer `x` remains minimal, due in part to its efficient reuse in CPU cache, which prevents us from measuring any performance penalty.

4.3 TVM integration in the HOPM benchmark

For the sake of completeness, this work also contemplates the higher-order power method (HOPM) [4], which is employed to find the best rank one approximation of a given tensor. Henceforth it is utilized as a benchmark since, given a tensor $\mathcal{A}$ of rank d, the algorithm performs $d(d-1)$ TVM operations across *all* tensor modes of $\mathcal{A}$. Therefore, the HOPM can be seen as an excellent tool to measure the *overall* performance of new TVM implementations.

Our sequential implementation requires up to two additional buffers to store intermediate tensors resulting from TVM operations. It also contains two in-house optimized functions `dot` and `axpby` that normalize the final output vectors. Since the first-touch based NUMA-aware algorithm outperforms the one using disjointed tensors, we achieve a NUMA-aware HOPM by the former approach.

TABLE 1: Number of 8-byte floating-point elements and memory footprint (within brackets) for the small (L3) and large (DDR) hypersquare tensors used in this work.

Order	L3 tensor (MB)	DDR tensor (GB)
2	1276^2 (13.0)	30001^2 (7.2)
3	116^3 (12.5)	989^3 (7.7)
4	34^4 (10.7)	157^4 (7.5)
5	16^5 (8.4)	62^5 (7.3)
6	10^6 (8.0)	31^6 (7.1)
7	7^7 (6.6)	19^7 (7.2)
8	5^8 (3.1)	13^8 (6.5)
9	4^9 (2.1)	9^9 (3.1)
10	4^{10} (8.4)	7^{10} (2.3)

If there are no memory constraints, one can allocate a *third* buffer to store a copy of the input tensor $\mathcal{A}$ with a different touch policy in relation to NUMA locality. The original tensor stored in buffer `a` is first-touched following the $(d-1)$-mode TVM memory access pattern and is later on employed on the HOPM for the 1-mode TVM over $\mathcal{A}$. At the same time, another copy of the original tensor is stored in buffer `aIT`, which has been previously first touched according to the 0-mode TVM memory access pattern, and is employed in every 0-mode TVM over $\mathcal{A}$ taking place during the HOPM operation. The two additional buffers used to store intermediate tensors do follow the same layout as the original buffer `a` since it yields minimal inter-NUMA communication for all modes except the first one, in which case the buffer `aIT` is used.

This HOPM implementation based on three temporary buffers maximizes performance at the expense of doubling the memory requirements for storing the input tensor $\mathcal{A}$. In NUMA systems with limited memory resources, as it can be the case for systems equipped with small high-bandwidth memories, it might be worth considering one single buffer for the input tensor with an inverse first-touch policy. In such scenarios, any performance degradation due to NUMA effects will only be bounded to the 1-mode TVM over $\mathcal{A}$ which occurs every $d(d-1)$ TVM operations, that is, once per HOPM external iteration.

5 Performance Evaluation

This section evaluates the performance of the TVM and HOPM native algorithms proposed in this work. From a hardware point of view, several architectures available at the Barcelona Supercomputing Center (BSC) are taken into account: Intel Xeon Platinum 8160, AMD EPYC 7742, Intel Xeon Phi 7230 (KNL) and Fujitsu ARM A64FX CPUs. From a software perspective, it is equally interesting to compare different compilers since the performance of the algorithms will ultimately depend on the generated binaries. To this end Intel, Clang and GCC compilers are considered.

Table 1 shows the tensors used in this work. These are hypersquare, dense tensors filled with 8-byte floating-point numbers, which is typical of this kind of studies [5], [9]. Taking the Intel Xeon Platinum 8160 CPU as a reference, two families of small and large tensors are considered. On the one hand, small tensors are selected so that all the associated working buffers fit within the 33 MiB of L3 cache memory. On the other hand, large tensors are constricted to occupy less than 8 GB so that all buffers used during a TVM or HOPM operation can be comfortably allocated in the 16 GB of high bandwidth memory (HBM) installed in the KNL. The length or size of each tensor dimension has been intentionally chosen to prevent it from being a multiple of the vector length (e.g. 8 when using doubles and 512-bit SIMD/SVE). However, the dimension size of 16 corresponding to the fifth-order L3 tensor is rather motivated by its memory footprint, which decreases monotonically with the tensor order. The tenth-order L3 tensor does not follow this rule because the minimum size per dimension is kept at 4. This particular election of dimension sizes presented in Table 1 is intended to maximize unaligned memory accesses. Similarly, the fact that large tensors are relatively small to fit in HBM, compared to other studies dealing with tensors of hundreds of gigabytes [9], means that the performance figures presented in this work correspond again to unfavourable scenarios. In this respect, much larger tensors with SIMD/SVE-friendly dimension sizes will certainly yield better performance figures.

With regard to compilation flags and taking the GCC compiler as a reference, `-march=native` is used to enable CPU specific optimizations, `-Ofast` combined with `-mprefer-vector-width=512` ensure that *full* 512-bit vector instructions are generated (except in the case of AMD, which remains a 256-bit vector architecture) and `-fopenmp` enables OpenMP pragma directives used for both vectorization and parallelization. Other code adjustments are made possible via preprocessor directives: L1 cache memory (e.g. `-DL1C=32768`, except for ARM which is 64 KiB), transparent huge pages memory (e.g. `-DTHP=2097152`), unroll factor (fixed at `-DUNROLL=8` at all times) and default vector length (`-DSIMD=512`, except for AMD which reduces to 256). Each benchmark runs a particular TVM or HOPM kernel during 5 seconds, which is enough to generate multiple kernels calls (between 10^2 and 10^6 instances) and extract statistical figures.

5.1 NUMA single-node experiments

5.1.1 L3 and DDR bandwidths

The first experiment conducted in this section is the popular STREAM benchmark. We utilize McCalpin's source code [19] and an in-house implementation written in C++ following the same optimization techniques included in our TVM algorithm. Contrarily to McCalpin's, the size of the buffers is not known at compile time in the in-house implementation. Both benchmarks are executed using all the 24 cores of the 8160 CPU and, for the sake of conciseness, only the `triad` function $\mathbf{z} := \mathbf{x} + k\mathbf{y}$ is considered.

Figure 3 reports on the *average* memory bandwidth for increasing values of the touched memory (the sum of buffers x, y and z) with various compilers. The L3- and DDR-bound scenarios are of great importance for the benchmarks that will be later shown. As expected, the peak bandwidth value above 1000 GB/s is reached at about half the L3 cache capacity. For very large buffers, all curves except one converge to an average bandwidth of about 80 GB/s. Indeed, the Intel compiler takes advantage of knowing the buffer size at compile time in McCalpin's *static* code and hence uses streaming stores on the output buffer `z`. These stores bypass

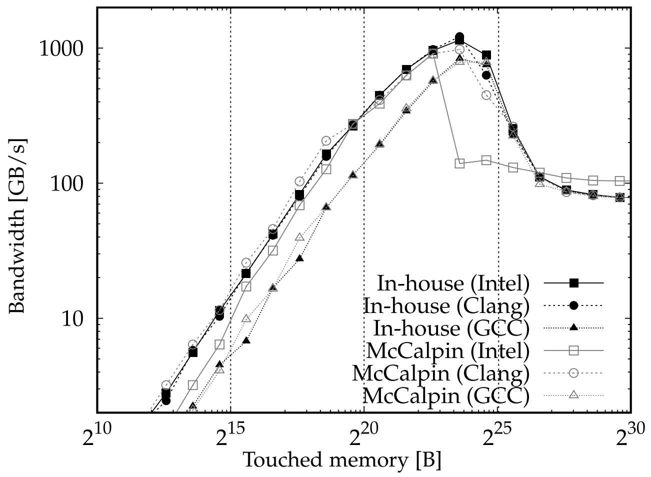


Fig. 3: STREAM `triad` function average bandwidth corresponding to an Intel Xeon Platinum 8160 CPU and various compilers. Black and gray colors refer to our in-house and McCalpin's [19] benchmark results, respectively. Vertical dotted lines correspond to L1 (32 KiB), L2 (1 MiB) and L3 (33 MiB) cache memory sizes of that CPU.

the CPU cache in order to write directly into system memory achieving 104.5 GB/s of bandwidth, a value that is closer to the theoretical 128 GB/s that the manufacturer reports for a single socket of MareNostrum 4 with two Intel CPUs.

We discuss other aspects of Fig. 3. First, the Intel compiler seems victim of its own success since it causes a premature performance drop for L3-bound buffers due to an early use of streaming stores. Second, GCC falls behind the other two compilers in terms of performance for L3-bound scenarios and smaller, especially when the buffer size is unknown at compile time. This lead us to believe that either the binaries generated by GCC or its OpenMP runtime are definitively not optimal (Intel and Clang compilers both use the same OpenMP runtime `KMP` while GCC has its own implementation). As a result, when the memory bottleneck moves from system RAM to CPU cache, it manifests in the form of overhead.

5.1.2 Native TVM algorithm performance

Table 2 shows the average bandwidth for the L3 and DDR tensors of Table 1. For a given tensor of rank d, this is in fact the average TVM performance over *all* contraction modes $k \in [0, 1, \ldots, d-1]$. The standard deviation over these modes is also specified in percentage terms. Best results correspond to greater average performance values (bold characters) and smaller deviation rates (in italics). Finally, the arithmetic mean of these two quantities over all tensors is reported on the last row in order to give a better idea of the average TVM performance for different compilers.

In an L3-bound scenario, Table 2(a), Intel reports the best results followed closely by Clang which is about 3% slower when looking at the arithmetic mean. At the same time, GCC is 30% slower than Intel. This performance regression affecting GCC coincides with the observations made on Fig. 3. Taking Intel results as a reference (although the same applies to the other compilers), the average TVM performance across modes varies significantly with the

TABLE 2: Average bandwidth (GB/s) and standard deviation percentage (within brackets) across all TVM contraction modes over the (a) small and (b) large tensors of Table 1. The arithmetic mean over all tensors ($\overline{\Sigma}$) is provided at the end for completeness. Bold and italic figures indicate best results in terms of performance and standard deviation, respectively. Results from different compilers on the Intel Xeon Platinum 8160 CPU.

(a) L3 tensor

Order	Intel	Clang	GCC
2	**787.6** (45.8)	721.6 (44.3)	675.9 (*19.9*)
3	**978.5** (8.5)	926.1 (7.4)	766.2 (*3.8*)
4	**869.3** (22.3)	845.5 (27.1)	667.8 (*19.9*)
5	937.4 (18.3)	**960.5** (20.0)	683.8 (*9.6*)
6	815.4 (41.5)	**822.7** (44.8)	565.2 (*34.2*)
7	**749.9** (*22.9*)	696.2 (35.0)	504.4 (29.7)
8	**585.5** (*22.3*)	576.3 (32.1)	338.8 (27.0)
9	**556.7** (*18.9*)	543.8 (30.0)	286.4 (21.5)
10	**1030.6** (*28.3*)	973.5 (38.6)	676.6 (35.7)
$\overline{\Sigma}$	**812.3** (25.4)	785.1 (31.0)	573.9 (*22.4*)

(b) DDR tensor

Order	Intel	Clang	GCC
2	107.4 (*0.3*)	**108.1** (*0.3*)	106.6 (1.9)
3	108.6 (3.1)	**108.8** (*2.9*)	106.1 (5.4)
4	**102.8** (11.3)	102.6 (11.6)	102.6 (*8.9*)
5	**99.4** (13.3)	**99.4** (13.4)	98.7 (*13.0*)
6	98.5 (10.9)	98.6 (11.0)	**98.7** (*8.5*)
7	93.1 (*10.8*)	**95.8** (11.6)	95.7 (11.6)
8	**94.2** (8.3)	93.6 (*7.7*)	93.8 (7.9)
9	**91.5** (9.8)	91.3 (*9.7*)	91.1 (*9.7*)
10	**95.0** (1.9)	94.4 (*1.8*)	94.8 (2.0)
$\overline{\Sigma}$	98.9 (7.8)	**99.2** (7.8)	98.7 (*7.7*)

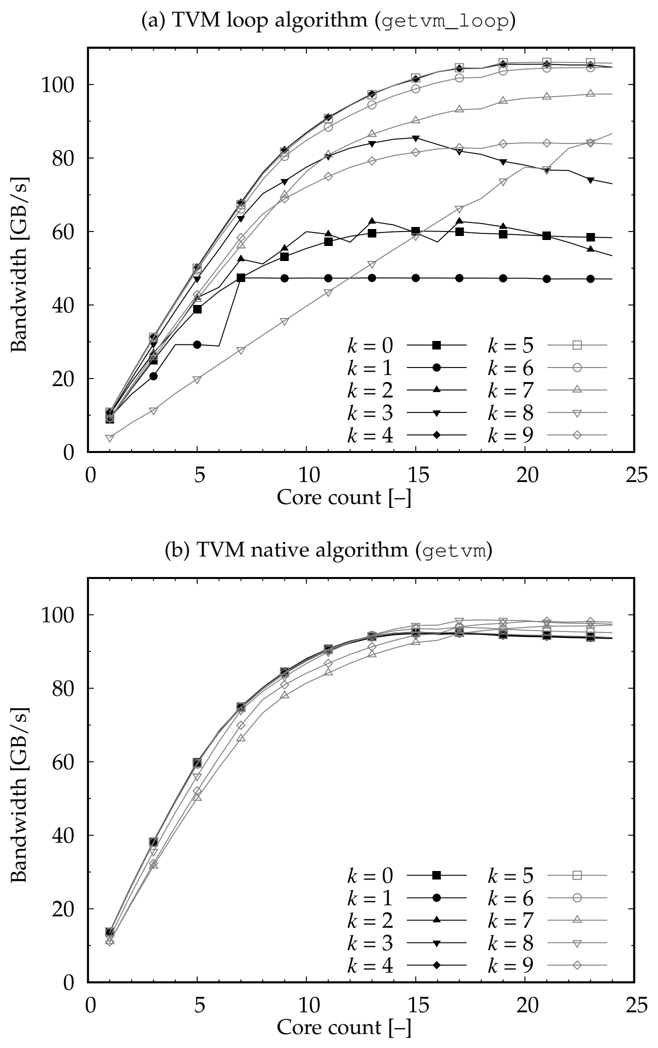


Fig. 4: k-mode TVM average bandwidth on the Intel Xeon Platinum 8160 CPU with increasing core count for (a) the getvm_loop algorithm based on Intel MKL gemv kernels and (b) the native algorithm getvm. Both binaries were generated by the Intel compiler. Results correspond to the hypersquare tensor composed of 7^{10} doubles of Table 1.

tensor order, which is expected of such tiny tensors. The average bandwidth per tensor is 812.3 GB/s and, although it may be deemed small in comparison to the peak value of 1165.4 GB/s measured in the STREAM benchmark with the same compiler, the tensors used here are indeed many times smaller than the L3 cache. Under these circumstances the parallel performance decreases with the touched memory, see Fig. 3, which ultimately explains that difference in performance. Lastly, the standard deviation rates reflect strong variations (of up to 45.8%) across modes, which is also expected of such small tensors. It is worth noting that it is mandatory to limit the vector length to 256 bits on this processor in order to get competitive L3 results over the last contraction mode $k = d - 1$. This is done via the preprocessor directive `-DSIMDMV=256`, which only adjusts the vector length of the kernels inside the function `matVec`.

When considering DDR-bound scenarios, Table 2(b), the results remain compiler agnostic. It can be readily seen that the performance decreases slightly with the tensor order and, considering GCC results in this case, the average bandwidth per tensor is 98.7 GB/s and fairly close to the peak bandwidth of 104.5 GB/s measured by STREAM bypassing CPU cache writes. The maximum standard deviation is only 13.4% with the average below 8%. These deviations are within the bounds reported by Pawłowski et al. [5] and therefore we can assert that this TVM algorithm is mode oblivious for DDR-bound dense tensors.

The TVM native algorithm is now compared against an in-house TVM loop algorithm (`getvm_loop`) that relies on the Intel MKL optimized kernel `gemv`. In order to maximize its parallel performance, this looped implementation combines the parallelism already integrated in the MKL function with an OpenMP "parallel for" directive at the outer loop over contiguous matrices. With the aim of keeping this part concise, only the arithmetic mean figures of the looped algorithm are reported here: 576.9 GB/s (43.2%) and 95.4 GB/s (17.4%) for the L3 and DDR tensors, respectively, using the same notation of Table 2. From these figures it can be inferred that the native algorithm is between $1.41\times$ and $1.03\times$ faster than its looped counterpart, on average, and also reduces the standard deviation rates by about half. A closer inspection at each contraction mode, not shown here for the sake of conciseness, reveals that the looped implementation remains mode-dependent in all scenarios

TABLE 3: HOPM average bandwidth (GB/s) for the (a) small and (b) large tensors of Table 1. Same legend as in Table 2. Results from different compilers on an Intel Xeon Platinum 8160 CPU.

(a) L3 tensor

O.	Intel	Clang	GCC
2	482.5	**489.1**	445.1
3	**590.4**	581.3	449.1
4	514.8	**537.5**	396.4
5	473.1	**478.2**	293.5
6	414.1	**417.0**	275.9
7	394.7	**406.9**	261.9
8	246.3	**256.4**	151.7
9	168.9	**171.5**	92.5
10	324.0	**327.6**	224.7
$\overline{\Sigma}$	401.0	**407.3**	287.9

(b) DDR tensor

O.	Intel	Clang	GCC
2	**106.9**	106.8	105.7
3	101.4	**102.4**	99.0
4	88.2	87.4	**93.0**
5	83.4	83.5	**83.7**
6	84.7	84.8	**88.8**
7	82.2	82.2	**82.4**
8	**84.9**	84.8	84.7
9	80.7	**80.8**	80.7
10	**92.3**	91.9	**92.3**
$\overline{\Sigma}$	89.4	89.4	**90.0**

and quickly begins losing performance with increasing tensor orders and contraction modes. This is clearly depicted in Fig. 4 for the tenth-order tensor with 7^{10} elements of Table 1. The axis of abscissae represents the core count to evidence any scalability issues present at each k-mode TVM. The results speak from themselves and demonstrate once more the mode-obliviousness properties of the native algorithm when applied to dense tensors stored following a non-hierarchical, last-order layout.

5.1.3 HOPM performance

The last benchmark to be run is the higher-order power method, which is another candidate for measuring the ultimate performance of a given TVM implementation. Table 3 shows the results corresponding to the HOPM with calls to `getvm` for the small (L3) and large (DDR) tensors executed on the Intel Platinum CPU. The reported bandwidths are smaller than those of Table 2 which is consistent with the HOPM benchmark performing consecutive TVM operations on successively smaller input tensors. As the touched memory involved in the HOPM decreases, so does its parallel performance. When that memory is smaller than the L1 cache size (32 KiB for this CPU), the corresponding kernels `getvm`, `dot` or `axpby` are executed sequentially for best throughput. Comparing the arithmetic mean results against those of Table 2, the HOPM bandwidth is halved with respect to the TVM bandwidth for L3 tensors while it is only reduced by less than 10% when considering large tensors. In the latter case, the benchmark is bounded by the system memory bandwidth, which is where all three compilers deliver similar performance. The HOPM standard deviation across DDR tensors stays below 10% on average, rendering the HOPM benchmark based on our native TVM algorithm oblivious to the tensor order.

To complete this section the HOPM benchmark based on calls to the TVM loop algorithm with Intel MKL optimized kernels (`gemv`, `dot` and `axpby`) is studied. The final results obtained by arithmetic mean are 166.9 GB/s and 68.1 GB/s for the small and large tensors, respectively. These figures clearly demonstrate the superiority of the native algorithm when integrated in the HOPM, making this benchmark between 2.4× and 1.31× faster than its looped counterpart.

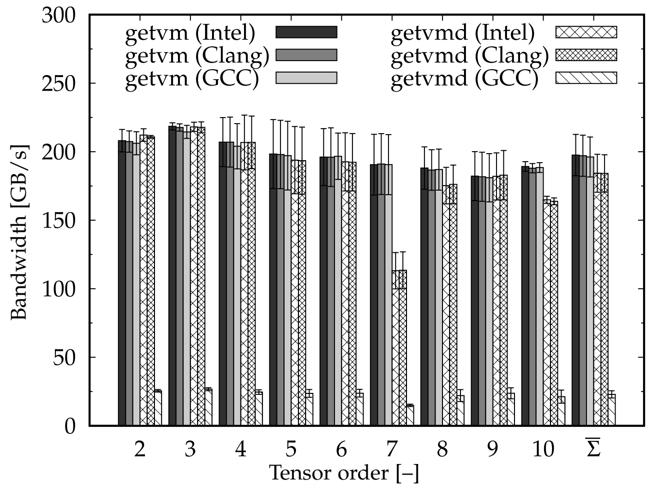


Fig. 5: Average bandwidth (GB/s) and standard deviation (error bars) across all TVM contraction modes over the large tensors of Table 1 provided by the `getvm` and `getvmd` algorithms. The arithmetic mean ($\overline{\Sigma}$) is also provided at the end for completeness. Results from different compilers on a two-socket system with Intel Xeon Platinum 8160 CPUs.

5.2 NUMA multi-node experiments

5.2.1 Two-socket system

The first NUMA experiments are carried out on a full compute node of MareNostrum 4 which consists of a two-socket system equipped with Intel Xeon 8160 CPUs with a theoretical bandwidth of 256 GB/s. However, McCalpin's STREAM `triad` function reports an average bandwidth of 155.5 GB/s and 205.6 GB/s when using the GCC and Intel compilers, respectively. This represents roughly 60% and 80% of the theoretical peak value and demonstrates again how the Intel binary benefits from using streaming stores in this particular benchmark.

Figure 5 reports on the TVM average bandwidth employing the two NUMA-aware strategies proposed in Section 4.2. Henceforth only the large tensors of Table 1 will be considered. By looking at the results corresponding to the TVM disjointed algorithm `getvmd`, it can be readily seen that GCC lacks support for OpenMP parallel nesting (two levels are required) at least up to version 10.1.0. Furthermore, this explicit NUMA-aware implementation somehow experiences performance regression on certain tensors. For instance, the measured bandwidth of the seventh-order tensor is barely better than that of the single-socket configuration, see Table 2(b). The same applies, but to a lesser extent, to the eighth- and tenth-order tensors.

On the other hand, Fig. 5 also reports on the TVM performance of the *implicit* NUMA-aware implementation provided by `getvm`. Since parallel nesting is not longer required, GCC is able to fully compete with Intel and Clang as evidenced by the mean bandwidth values. Indeed, the three compilers deliver once again similar performance. For lower-order tensors the measured bandwidths are slightly larger than those reported by STREAM. What is more, the results also indicate that the standard deviation remains invariant of the number of NUMA nodes and stays, on average, below 8% while the speedup reaches 1.99×.

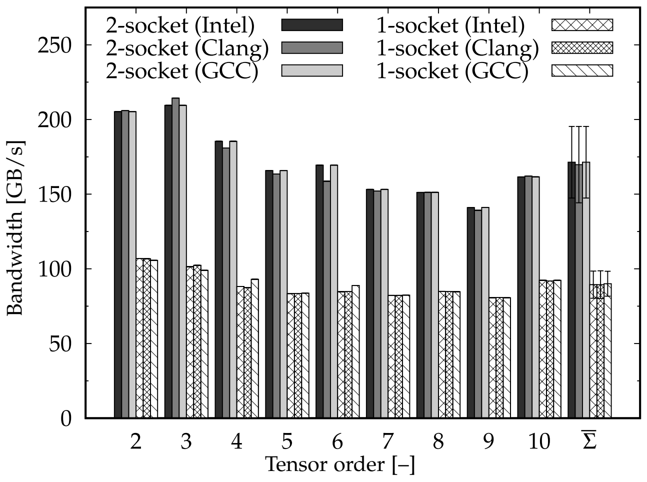


Fig. 6: HOPM average bandwidth (GB/s) for the large tensors of Table 1. The arithmetic mean ($\overline{\Sigma}$) is also provided at the end for completeness. Results from different compilers on one- and two-socket systems equipped with Intel Xeon Platinum 8160 CPUs.

All things considered, `getvm` is 7% faster than `getvmd` and manages to attain more than 76% of the theoretical bandwidth when considering the arithmetic mean over all tensors. The implicit NUMA-aware implementation of `getvm`, which is fully supported by the GCC OpenMP runtime, is faster and much simpler than its disjointed counterpart. Hence, from now on, we study only the implicit NUMA-aware `getvm` version.

Figure 6 reports on the HOPM average bandwidth based on calls to `getvm` for this two-socket system. The three compilers are tied and the arithmetic mean figures reflect an overall speedup of $1.9\times$ over the single-socket results. Although the HOPM bandwidth results remain competitive, these figures are on average about 10% inferior to those reported by the TVM algorithm; indeed, the HOPM benchmark attains more than 66% of the theoretical bandwidth compared to the 76% of the TVM. This is again attributed to the `getvm`, `dot` or `axpby` functions whose memory footprint is smaller than the L1 cache and hence are executed sequentially within the HOPM. Although this results in overall best performance, it does not necessarily yield best NUMA scalability and therefore we cannot longer claim that the HOPM performance for more than one NUMA node is oblivious to the tensor order.

5.2.2 Single-socket systems

The AMD Epyc 7742, Intel Xeon Phi 7230 (KNL) and Fujitsu ARM A64FX CPUs are now tested. The first one is made of chiplets and uses conventional DDR while the other two benefit from HBM. These three CPU architectures are composed of 4 NUMA nodes and therefore it is extremely important to setup their firmware in such a way that all the nodes are available to the programmer in order to minimize inter-NUMA communication as much as possible.

In terms of compilers, if one considers performance scenarios limited by the system memory (DDR or HBM), as it is the case of this section, then GCC delivers the best

TABLE 4: Average system memory bandwidth, as measured by McCalpin's STREAM `triad` function [19], and theoretical bandwidth for an increasing number of NUMA nodes on various hardware architectures: AMD Epyc 7742, Intel Xeon Phi 7230 (KNL) and ARM A64FX. The smaller figures reported by STREAM were obtained via GCC binaries which do not bypass CPU cache writes. Units are GB/s.

Platform	1-NUMA	2-NUMA	4-NUMA
AMD	27.7/ 51.2	55.2/102.4	110.2/ 204.8
KNL	84.4/115.2	168.1/230.4	331.7/ 460.8
ARM	153.7/256.0	306.7/512.0	609.8/1024.0

results on the ARM architecture while being just as competitive as other compilers (vendor or Clang) on the other three architectures. For this reason, we employ this free and open-source compiler herein to bring the best comparison between the four platforms. GCC version 10.1.0 is available for both the AMD Epyc and Intel KNL CPUs (as well as the Intel Platinum CPU). For the ARM platform, we utilize an *experimental* beta version of GCC compatible with SVE and based on the 11.0.0 beta release.

Table 4 shows the average system memory bandwidth measured via McCalpins STREAM benchmark compiled with GCC as well as the theoretical bandwidth according to the hardware manufacturer for the three platforms. Different values are presented depending on the number of NUMA partitions. Considering the last case with 4 nodes, the measurements represent roughly 53%, 72% and 59% of the theoretical bandwidth of the AMD, KNL and ARM platforms, respectively. This is because the corresponding GCC binaries do not use streaming stores. As it was the case for the Intel Platinum CPU, one should expect TVM and HOPM performance figures ranging from the values measured via STREAM and the theoretical ones reported on Table 4.

Table 5 summarizes the average parallel performance of the TVM native algorithm for different architectures and NUMA partitions. Starting with the AMD platform, Table 5(a), it can be noted that the performance scales pretty well with the number of nodes. The accumulated standard deviation percentages are similar to those previously reported on the Intel Platinum CPU which also utilizes conventional DDR. Looking at the 4-NUMA results, the average bandwidth per tensor is 136.2 GB/s, which is above the 110.2 GB/s reported by STREAM. The TVM algorithm reaches, on average, 66% of the theoretical bandwidth on the AMD platform. This is about 10% less than what was achieved on the previous two-socket Intel system with monolithic CPUs.

The effects of using fast RAM —HBM in the case of the KNL and HBM2 for the ARM CPU— are shown on Table 5(b)–(c). For example, it can be readily seen that the standard deviation rates obtained by arithmetic mean increase by almost a factor of two compared to the same values reported on the previous two systems with conventional DDR. A closer inspection reveals that mode obliviousness on higher-order tensors cannot longer be considered for the KNL. This is even worse for the ARM architecture since the 0-mode TVM clearly indicates that there is a major perfor-

TABLE 5: Average bandwidth (GB/s) and standard deviation percentage (within brackets) across all TVM contraction modes over the large tensors of Table 1. Same legend as in Table 2. Results provided by the GCC compiler for an increasing number of NUMA nodes on (a) AMD Epyc 7742, (b) Intel Xeon Phi 7230 (KNL) and (c) ARM A64FX CPUs.

(a) AMD

Order	1-NUMA	2-NUMA	4-NUMA
2	38.2 (1.3)	75.3 (2.6)	144.5 (5.5)
3	36.4 (10.0)	75.8 (1.8)	151.5 (1.4)
4	36.5 (9.0)	72.8 (8.9)	144.3 (8.2)
5	33.9 (12.7)	67.8 (12.6)	134.7 (12.1)
6	34.7 (8.3)	69.3 (8.3)	137.7 (7.8)
7	32.9 (11.0)	65.7 (10.9)	130.7 (10.7)
8	32.8 (7.7)	65.6 (7.7)	130.1 (7.3)
9	31.7 (9.0)	63.3 (9.0)	124.5 (8.9)
10	32.6 (1.7)	65.0 (1.6)	127.8 (1.5)
$\overline{\Sigma}$	34.4 (7.9)	69.0 (7.0)	136.2 (7.0)

(b) KNL

Order	1-NUMA	2-NUMA	4-NUMA
2	91.1 (3.3)	179.7 (5.0)	329.2 (14.9)
3	93.8 (2.3)	184.1 (3.8)	355.2 (5.5)
4	87.0 (6.7)	170.4 (4.7)	334.5 (4.1)
5	81.3 (15.4)	155.5 (12.3)	279.5 (9.3)
6	83.9 (21.4)	159.9 (19.6)	289.4 (20.0)
7	84.2 (20.4)	158.1 (18.4)	278.9 (15.0)
8	86.0 (18.6)	161.7 (16.3)	275.6 (21.3)
9	86.8 (23.1)	154.8 (18.7)	303.7 (20.0)
10	82.0 (23.9)	162.6 (24.3)	317.0 (24.5)
$\overline{\Sigma}$	86.2 (15.0)	165.2 (13.7)	307.0 (14.9)

(c) ARM

Order	1-NUMA	2-NUMA	4-NUMA
2	182.4 (27.4)	344.7 (37.1)	616.0 (55.6)
3	171.0 (13.7)	334.7 (16.3)	655.8 (18.6)
4	146.4 (10.8)	291.2 (10.3)	577.9 (9.8)
5	154.4 (14.0)	308.2 (14.0)	612.8 (13.8)
6	161.3 (10.1)	320.5 (9.7)	635.1 (9.3)
7	173.1 (10.6)	345.5 (10.6)	688.0 (10.4)
8	169.5 (12.6)	338.3 (12.6)	674.4 (12.6)
9	151.6 (19.8)	302.6 (19.7)	603.9 (19.3)
10	168.1 (18.6)	335.8 (18.6)	672.2 (17.9)
$\overline{\Sigma}$	164.2 (15.3)	324.6 (16.6)	637.3 (18.6)

TABLE 6: HOPM average bandwidth (GB/s) for the large tensors of Table 1. Same legend as in Table 2. Results provided by the GCC compiler for an increasing number of NUMA nodes on (a) AMD Epyc 7742, (b) Intel Xeon Phi 7230 (KNL) and (c) ARM A64FX CPUs.

(a) AMD

O.	1-N.	2-N.	4-N.
2	38.2	75.3	144.1
3	34.1	75.1	147.9
4	33.1	66.0	130.7
5	29.4	58.4	115.7
6	31.3	61.9	121.9
7	29.1	57.1	112.0
8	29.9	57.8	109.4
9	28.2	54.6	102.3
10	32.3	62.0	111.0
$\overline{\Sigma}$	31.7	63.1	121.7

(b) KNL

O.	1-N.	2-N.	4-N.
2	73.9	125.6	323.5
3	69.7	112.5	349.8
4	68.8	111.2	339.6
5	65.7	109.2	326.1
6	79.1	116.6	332.8
7	74.1	112.1	254.7
8	82.2	132.4	317.7
9	97.1	160.6	293.7
10	110.8	180.2	324.0
$\overline{\Sigma}$	80.2	128.9	318.0

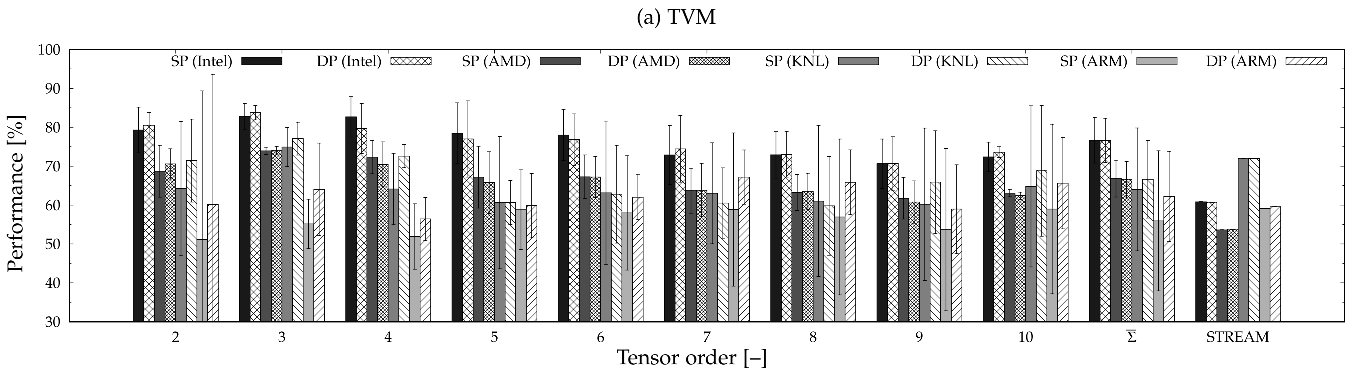
(c) ARM

Order	1-NUMA	2-NUMA	4-NUMA
2	151.1	320.6	515.8
3	128.0	293.4	570.0
4	127.0	311.9	615.0
5	135.1	336.4	657.9
6	135.7	334.5	637.5
7	134.1	334.6	631.3
8	148.9	340.6	628.5
9	165.5	313.2	544.0
10	181.4	341.8	563.6
$\overline{\Sigma}$	145.2	325.2	595.9

mance gap of up to 55.6% between the VMM and MVM kernels. We believe that this may be due to the binaries generated by the experimental version of GCC installed in this platform. On the other hand, the NUMA scalability remains pretty good, especially on ARM platform. The KNL delivers an average bandwidth per tensor of 307 GB/s, that is about 66% of the theoretical value and slightly below the one reported by STREAM. On ARM this value is 637.3 GB/s, just above the STREAM measurement, and represents 62% of the theoretical bandwidth. All things considered, these three architectures exhibit similar performance figures in relative terms and remain below the average value reported on the previous two-socket Intel platform (see Fig. 7(a)). This is perhaps an indication that more sophisticated hardware architectures, equipped or not with high bandwidth memory, become more challenging to fully exploit. In this regard, compilation parameters such as the unroll factor have remained unchanged for all cases and hence no additional efforts have been made to fine-tune the compilation for a particular architecture.

Finally, Table 6 summarizes the average bandwidth achieved by the HOPM native algorithm for different architectures and NUMA partitions. One may expect that the HOPM performance would always stay below the TVM performance but, surprisingly, this is not the case for the KNL using 4 NUMA nodes and the ARM CPU with 2 nodes. The arithmetic mean figures corresponding to the 4-NUMA case indicate that the HOPM reaches, on average, 59%, 69% and 58% of the theoretical bandwidth on the AMD, KNL and ARM processors, respectively. In relative terms, the KNL performs slightly better than the two-socket Intel platform. For the AMD and ARM architectures, the approximate 10% performance drop-off observed in the TVM algorithm is carried forward in the HOPM benchmark.

The results presented in this section have demonstrated the performance improvements brought by the TVM native algorithm in various scenarios. All the mathematical operations have been carried out using double precision arithmetic over the dense tensors described in Table 1. Figure 7 presents the average performance of the TVM and HOPM for the same tensors filled with single precision floating-point numbers when using all the NUMA nodes available on the four architectures considered in this work. Note that since the number of elements per tensor does not change, their memory footprint is reduced by half. In order to



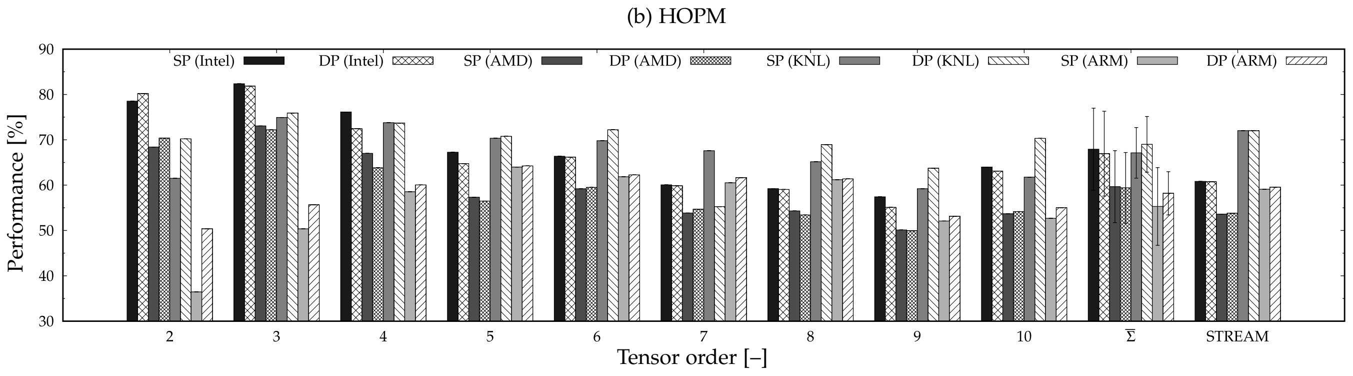


Fig. 7: Single and double precision average performance (relative to the theoretical bandwidth) and standard deviation (error bars) across all TVM contraction modes for the large tensors of Table 1 provided by the (a) TVM algorithm and (b) HOPM benchmark. The arithmetic mean over all tensors ($\overline{\Sigma}$) and the STREAM performance figures are also provided at the end for completeness. Results correspond to the four architectures considered in this work exploiting their full potential.

facilitate the comparison between different architectures the performance is expressed as a percentage of the theoretical bandwidth. Similarly, double precision performance figures previously obtained are included in the comparison.

Figure 7(a) shows that the TVM performance is invariant of the arithmetic precision for Intel and AMD processors, especially when looking at the arithmetic mean values. There is a slight reduction of bandwidth on the KNL CPU when using 4-byte floats and a definitive performance gap on the ARM architecture. We take the ARM results with a grain of salt since we have experienced inconsistent behaviour over certain contraction modes where the multi-threaded performance was barely better than the single-threaded one; it may be the experimental version of the GCC compiler not being able to generate proper SVE vector instructions in this particular scenario. It is worth mentioning that the principal consequence of reducing the floating-point precision is an overall increment of the standard deviation rates of 69% and 73% for the KNL and ARM architectures, respectively. We attribute this to the HBM installed in these systems.

Finally, Figure 7(b) shows that, on average, the HOPM performance also tends to decrease for the KNL and ARM CPUs due to the use of HBM. In this particular case, the performance drop is of the same order of magnitude for the two architectures. However, only the standard deviation value across all tensors increases by almost a factor of two on the ARM CPU when using single precision arithmetic, while it remains practically unaltered on the KNL system.

6 Conclusion and future work

This work has presented a novel tensor–vector multiplication (TVM) algorithm in the form of an exclusive BLAS level 2 function. It has been built from scratch, uses a non-hierarchical layout, contains specialized optimizations and shares the same list of arguments as the well-known BLAS matrix–vector multiplication function plus an additional term. Moreover, its multi-threaded implementation is straightforward and, because it presents a flat parallel structure, delivers better scalability and performance over other popular approaches such as the TVM unfold and loop algorithms. Two first-touch strategies have been proposed with the objective of bringing NUMA awareness to TVM although working with disjointed tensors has proven to yield worse results. Lastly, this TVM native algorithm has been integrated in the higher-order power method (HOPM) that serves as a real-world benchmark.

The TVM and HOPM algorithms have been satisfactorily tested using up to three different compilers and four hardware architectures, from which two of them feature high bandwidth memory. For large tensors, the performance results are limited by the system memory bandwidth and remain compiler agnostic. Moreover, mode obliviousness is achieved on architectures with conventional DDR contrarily to what is observed in the case of the TVM loop algorithm, even when relying on heavily optimized BLAS libraries with architecture-tailored kernels. However, the proposed TVM exhibits some mode dependency in the presence of HBM since the associated TVM deviation rates tend to double. In general, the bandwidth achieved by the proposed algorithm oscillates between 62% and 76% for the TVM and from 58% to 69% for the HOPM relative to the theoretical system memory speeds reported by the manufacturers. These percentages diminish slightly on systems equipped with HBM when performing single precision arithmetic computations. Our results also confirm that the overall performance tends to diminish as a consequence of using more complex architectures composed of several NUMA partitions.

Although all the figures shown in the present study are related to worst-case scenarios due to the particular selection of tensors, they remain very competitive and clearly assess the effectiveness of the proposed TVM algorithm, making it suitable for integration in the most popular BLAS libraries available today. Future work will be carried out to integrate task-based shared memory programming models as well as the message passing interface (MPI) for distributed parallelism. Porting the TVM library to GPUs and other accelerators is also envisaged. Finally, the fundamental principles applied to our TVM algorithm will also be extended to the BLAS level 3 tensor–matrix multiplication algorithm and, by extension, to the more general tensor–tensor multiplication operation.

Acknowledgments

This work has been developed with the support of MCIN/AEI/10.13039/501100011033 and ESF/10.13039/501100004895 under Grant RYC2019-027592-I and has also benefited financially from the HPC Technology Innovation Lab, a Barcelona Supercomputing Center and Huawei research cooperation agreement (2020).

References

[1] W. Greub, *Multilinear Algebra*. Springer New York, 1978.

[2] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "Tensors for data mining and data fusion: models, applications, and scalable algorithms," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 2, 2016.

[3] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[4] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "On the best rank-1 and rank-(R1,R2,. . . ,RN) approximation of higher-order tensors," *SIAM J. Matrix Anal. Appl.*, vol. 21, no. 4, pp. 1324–1342, 2000.

[5] F. Pawłowski, B. Uçar, and A. N. Yzelman, "A multi-dimensional Morton-ordered block storage for mode-oblivious tensor computations," *Journal of Computational Science*, vol. 33, pp. 34–44, 2019.

[6] P. Springer and P. Bientinesi, "Design of a high-performance GEMM-like tensor–tensor multiplication," *ACM Trans. Math. Softw.*, vol. 44, no. 3, Jan. 2018.

[7] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs," in *SC '13: Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[8] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka, "Tensor contractions with extended BLAS kernels on CPU and GPU," in *2016 IEEE 23rd Int. Conf. on High Performance Computing (HiPC)*, 2016, pp. 193–202.

[9] F. Pawłowski, B. Uçar, and A. N. Yzelman, "High performance tensor–vector multiplication on shared-memory systems," in *Parallel Processing and Applied Mathematics*. Springer International Publishing, 2020, pp. 38–48.

[10] E. Di Napoli, D. Fabregat-Traver, G. Quintana-Ort, and P. Bientinesi, "Towards an efficient use of the BLAS library for multilinear tensor contractions," *Applied Mathematics and Computation*, vol. 235, pp. 454–468, 2014.

[11] C. Bassoy, "Design of a high-performance tensor-vector multiplication with BLAS," in *Computational Science – ICCS 2019*. Springer International Publishing, 2019, pp. 32–45.

[12] "Intel® oneAPI math kernel library developer reference," last visited May 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/mkl-reference-manual.html

[13] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "LIBXSMM: Accelerating small matrix multiplications by runtime code generation," in *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016.

[14] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, pp. 14:1–14:33, June 2015.

[15] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, pp. 77:1–77:29, Oct. 2017.

[16] D. A. Matthews, "High-performance tensor contraction without transposition," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C1–C24, Jan. 2018.

[17] P. J. Martinez-Ferrer, "TVM library [Computer software]," 2021. [Online]. Available: https://doi.org/10.24433/CO.9368326.v1

[18] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth, "Challenges of memory management on modern NUMA systems," *Commun. ACM*, vol. 58, no. 12, pp. 59–66, 2015.

[19] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pp. 19–25, 1995.

Dr. Pedro J. Martinez-Ferrer received two engineering degrees in aeronautical engineering at Universidad Politcnica de Madrid (UPM) and Ecole Nationale Suprieure de Mcanique et d'Arotechnique (ENSMA) in 2010 and a PhD at ENSMA in 2013. He has a strong experience in aeronautical and software engineering accumulated in several European research institutions and universities in Norway (SINTEF), France (Institut PPRIME), United Kingdom (MMU) and Spain (BSC). He has participated in EPSRC computational collaborative projects as well as the DEEP-EST European project. He is currently a postdoctoral researcher at Universitat Politècnica de Catalunya (UPC) and Ramón y Cajal fellow.

Dr. Albert-Jan Nicholas Yzelman is a research scientist with the Computing Systems Laboratory at Huawei Technologies Switzerland, Zürich Research Center. He obtained his M.Sc. and Ph.D. degrees in Mathematics in 2007 and 2011 from Utrecht University, the Netherlands. Yzelman previously held positions as senior and principal researcher at the Huawei Paris Research Center, France, and held a post-doctoral position at the KU Leuven and the Flanders ExaScience Lab in Leuven, Belgium. His research interests include programming models, parallel algorithm design, high performance computing, irregular computations and hypergraph partitioning.

Dr. Vicenç Beltran received his Engineering and Ph.D. degrees in Computer Science in 2004 and 2009, both from the Technical University of Catalonia. Since 2009, he is Senior Researcher at the Barcelona Supercomputing Center, where he works on parallel and distributed programming models, hardware accelerators, domain specific languages and operating systems. He has participated in several projects, including DEEP (leading the WPs on programming models and resiliency), INTERTWinE (leading the WP on runtime interoperability) and REPSOLVER (leading the development of a DSL infrastructure). He currently leads the Runtime Systems for Parallel Programing Models group that develops the OmpSs-2 programming model.