A multi-dimensional Morton block storage for mode-oblivious tensor computations

F. Pawłowski[1,2] B. Uçar[2] A.J. Yzelman[1]

[1]Huawei Technologies France
20 Quai du Point du Jour, 92100 Boulogne-Billancourt, France

[2]ENS Lyon, France

1 March 2019

Motivation

Examples of tensors:

- Dense: Images and videos
- Sparse: User-product-time database of an online store

Tensor algorithms help analyze data

They rely on these core tensor kernels:

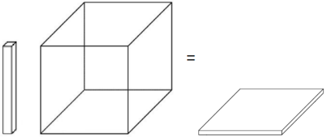
- Tensor–vector multiplication (*TVM*)
- Tensor–matrix multiplication (*TMM*)
- Khatri–Rao product

Motivation: *TVM*

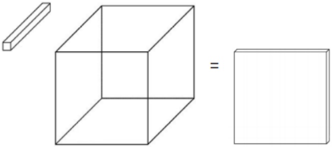
Tensor–vector product is denoted with the symbol $\overline{\times}_k$:

$$\mathcal{P} = \mathcal{A} \; \overline{\times}_k \; v, \quad \mathcal{P} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_{k-1} \times 1 \times n_{k+1} \times \cdots \times n_d}$$

TVM can be applied along any of the modes:



$\mathcal{A} \; \overline{\times}_1 \; v$



$\mathcal{A} \; \overline{\times}_3 \; v$

Contribution

We propose a blocked storage for dense tensors:

Morton-order on blocks,
regular ordering within blocks

We evaluate our method against the state-of-the-art *TVM*

The proposed storage is mode-oblivious on a TVM kernel

Related work

Li et al. (2015) discuss an algorithm for *TMM* using BLAS3 routines and an auto-tuning approach

Li et al. (2018) apply Morton-order for sparse tensors

Lorton and Wise (2007) apply Morton-order for dense matrices

Kjolstad et al. (2017) propose taco, a tensor algebra compiler (code generator) for tensor computations

Tensor layout

Tensor **layout** maps tensor elements $\mathcal{A}_{i_1,\ldots,i_d}$ onto an array of size $N = \Pi_{i=1}^{d} n_i$ and is denoted by $\rho(\mathcal{A})$:

$$\{1, \ldots, n_1\} \times \cdots \times \{1, \ldots, n_d\} \mapsto \{1, \ldots, N\}.$$

Describes the order in which the tensor elements are stored

Unfold layout ρ_π (multidimensional array)

Morton layout ρ_Z

Unfold layout ρ_π

Multidimensional array storage

Associated with a permutation π of $(1, \ldots, d)$

Tensor stored according to one of $d!$ orderings π of the modes

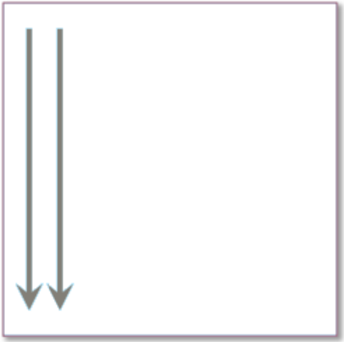
Convention: the rightmost mode in the permutation is one whose index is the fastest changing in memory

TVM reduces to matrix–vector multiplication (*MVM*) kernels

Unfold layout ρ_π: Matrix

There are two possible storages for a matrix (a tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2}$):

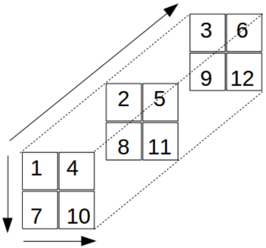
$\pi = (1, 2)$



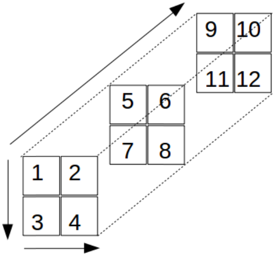
$\pi = (2, 1)$

Unfold layout ρ_π: 3D Example

There are 6 (3!) ways of storing an order-3 tensor:



$\pi = (1, 2, 3)$



$\pi = (3, 1, 2)$

$\dots$

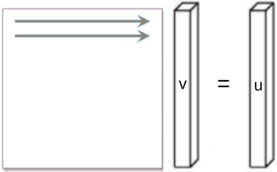
Example:
$\rho_\pi(\mathcal{A})(2, 2, 2) = 11$
$\rho_\pi^{-1}(\mathcal{A})(8) = (2, 1, 2)$

TVM using unfold layout: DGEMV kernels

We formulate *TVM* in terms of *MVM* BLAS2 kernels

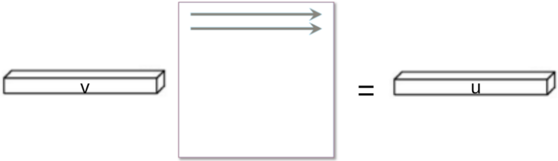
Both assume a $\rho_{(1,2)}$ layout for A

2 available BLAS2 *MVM* kernels:



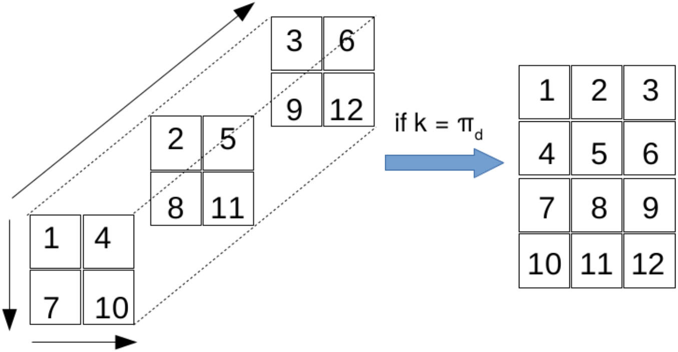
mv kernel: the right-hand sided multiplication ($u = Av$)

vm kernel: the left-hand sided multiplication ($u = vA$)



TVM using unfold layout: Matricization

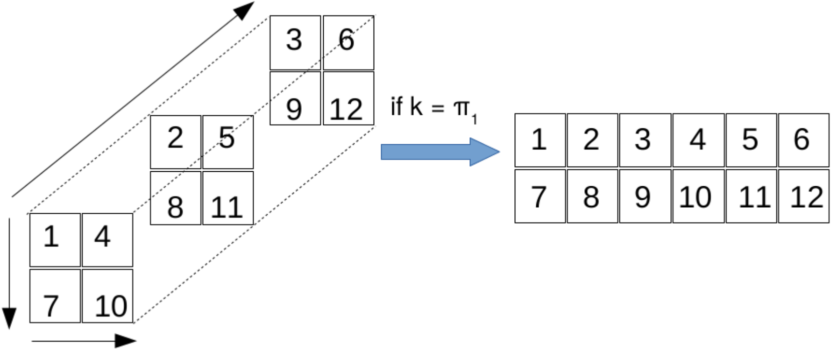
Mode d: apply a single mv on a tensor matricized as a tall-skinny $N/n_d \times n_d$ matrix



$\mathcal{A} \;\overline{\times_3}\; v$ reduces to single $(n_1 n_2 \times n_3)$ mv

TVM using unfold layout: Matricization

Mode 1: apply a single *vm* on a tensor matricized as short-wide $n_1 \times N/n_1$ matrix



$\mathcal{A} \ \overline{\times_1} \ v$ reduces to single $(n_1 \times n_2 n_3)$ *vm*

TVM using unfold layout: State-of-the-art

For the other $d - 2$ modes there are two algorithms:

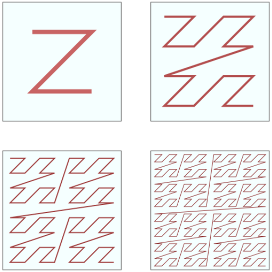
tvUnfold (transpose-DGEMV) explicitly rearranges tensor memory into a $\pi = (k, \ldots)$ tensor,
aligned for a single $(n_k \times N/n_k)$ *vm* kernel

tvLooped (loop-over-DGEMV)
runs N/r executions of $(n_k \times r/n_k)$ *vm* kernels

Morton layout ρ_Z

Morton-order is an ordering with locality preserving properties

Typically implemented using binary permutations



TVM using Morton layout: Optimized implementation

Array mortonResultIndex of size $\lceil \log max(n_k) \rceil$ one-initialized

Array *coord* of current element coordinates in the tensor

for $tensorIndex = 1$ to N **do**
    $\mathcal{B}_{resultIndex}$ += $\mathcal{A}_{tensorIndex} v_{vectorIndex}$
    $resultIndex \leftarrow resultIndex + 1$
    level, offset $\leftarrow$ *incMortonCoord(coord)*
    if $offset = k$ **then**
        swap(mortonResultIndex[level], resultIndex)
        blockDiff $= ceil(\log_2(max_k(n_k) - coord_k))$
        if blockDiff $<$ level **then**
            level $=$ blockDiff
        end if
        for $j = 1$ to *level* **do**
            mortonResultIndex[j] $=$ resultIndex
        end for
    end if
    $vectorIndex = coord_k$
end for

TVM using Morton layout: Complexity

Space complexity: $\Theta(d + log_2 n)$

Time complexity:

1. Line 3: *incMortonCoord*: $\Theta(N)$
2. Lines $4 - 13$:

$$\Theta\left(\sum_{i=0}^{log_2 n-1} 2^{di+k} + \sum_{i=0}^{log_2 n-2} 2^{di+log_2 n-2+k}\right) = \Theta(N/2^{d-k})$$

Block *TVM* algorithms

We propose Morton-blocked storage $\rho_Z \rho_\pi$

with smaller equally-sized tensors as its elements

and blocks stored as ρ_π to use BLAS2 kernels

We implemented

the associated $\rho_Z \rho_\pi$ *TVM* algorithm

the $\rho_\pi \rho_\pi$ *TVM* algorithm for comparisons

Complexity drops from N to N/B, where B is the block size

Setup

- Intel Ivy Bridge node with two Intel Xeon E5-2690 v2 processors (10 cores each)
- We measure sequential execution
- The processor has 32 KB of L1 cache, 256 KB of L2 cache memory, and 25 MB of L3 cache memory
- We assume square tensors and blocks
- Tensors of several GB
- We benchmark for $d = 2$ tensors up to $d = 10$

Implementation

MKL library for unblocked tensors

MKL and LIBXSMM [1] for individual blocks

MVM kernel performance depends on size of the matrix, its aspect ratio as well as its orientation

Microbenchmarks indicate that L2 and L3 block size yields best performance, in particular 2.5MB (10% of L3)

Microbenchmark results

TVM is bandwidth-bound due to low arithmetic intensity, between 1 and 2:

$$\frac{2\prod_{i=1}^{d} n_i}{\prod_{i=1}^{d} n_i + \frac{\prod_{i=1}^{d} n_i}{n_k} + n_k} \; \textit{flop per element} \, . \tag{1}$$

STREAM benchmark at 18.3 GB/s

Copy routine at 11.4 GB/s

Comparisons (effective bandwidth)

d	taco	*tvUnfold*	*tvLooped*	$\rho_\pi \rho_\pi$-block	$\rho_Z \rho_\pi$-block
2	9.36	12.22	12.22	14.09	**14.10**
3	11.92	6.36	**12.47**	10.90	11.06
4	10.09	4.50	10.79	11.77	**11.86**
5	10.69	3.76	10.71	12.03	**12.06**
6	9.93	3.46	10.98	11.20	**11.48**
7	9.55	3.64	11.26	10.69	**11.52**
8	6.94	3.68	**11.13**	9.28	10.87
9	6.75	3.54	**10.82**	8.66	10.36
10	7.05	3.77	10.26	9.14	**10.62**

Table: Average effective bandwidth (in GB/s) of different algorithms for large order-d tensors. The highest bandwidth, signifying the best performance, for each d is shown in **bold**. Tensor sizes n are such that at least several GB of memory is required.

Comparisons (standard deviation between modes)

d	taco	*tvUnfold*	*tvLooped*	$\rho_\pi\rho_\pi$-block	$\rho_Z\rho_\pi$-block
2	18.50	6.66	6.66	1.15	**0.65**
3	38.25	83.75	20.24	14.31	**12.99**
4	38.18	80.03	**6.27**	9.78	10.31
5	33.65	77.73	13.49	8.47	**7.08**
6	30.30	88.34	11.07	16.54	**8.58**
7	28.50	81.39	10.07	24.79	**4.73**
8	11.53	75.93	12.29	27.49	**5.82**
9	10.21	77.26	14.98	35.82	**9.44**
10	11.03	74.71	18.56	33.79	**9.17**

Table: Relative standard deviation (in percentage, versus the average bandwidth) of different algorithms for large order-d tensors. The lowest standard deviation, signifying the best mode-oblivious behavior, for each d is shown in **bold**. Tensor sizes n are such that at least several GB of memory is required.

Case study: Higher order power method

High-order power method is used to find rank-one tensor approximation

for *iters* = *0* to *maxIters* -1 **do**
  for $k = 0$ to $d-1$ **do**
    $\tilde{u}^{(k)} \leftarrow \mathcal{A}$
    for $t = 0$ to $k-1$ **do**
      $\tilde{u}^{(k)} \leftarrow \tilde{u}^{(k)} \ \overline{\times}_t \ u^{(t)}$
    end for
    for $t = k+1$ to $d-1$ **do**
      $\tilde{u}^{(k)} \leftarrow \tilde{u}^{(k)} \ \overline{\times}_t \ u^{(t)}$
    end for
    $u^{(k)} \leftarrow \frac{\tilde{u}^{(k)}}{\|\tilde{u}^{(k)}\|}$
  end for
end for
return $(u^{(0)}, u^{(1)}, \ldots, u^{(d-1)})$

Comparisons (effective bandwidth)

d	*tvLooped*	$\rho_\pi \rho_\pi$-block	$\rho_Z \rho_\pi$-block
2	11.10	13.96	**13.98**
3	**13.99**	9.85	9.80
4	9.64	**11.32**	11.29
5	9.83	12.80	**12.82**
6	10.88	**12.65**	12.63
7	10.90	12.47	**12.50**
8	10.82	12.34	**12.35**
9	10.30	11.74	**11.76**
10	9.69	11.42	**11.46**

Table: Average effective bandwidth (in GB/s) of different algorithms for HOPM of large order-d tensors on an Intel Ivy Bridge node. The highest bandwidth, signifying the best performance, for each d is shown in **bold**. Tensor sizes n are such that at least several GBs of memory is required.

Conclusions

The preferred method is the $\rho_Z\rho_\pi$-block *TVM* algorithm:

- Mode-oblivious performance with standard deviations $\leq 10\%$ (except for $d = 3$)
- Better performance

Transfers to other architectures, with significant speedups on both Ivy Bridge and Haswell nodes

Future work

- Other operations: *TMM* and Khatri–Rao product
- Parallel implementations
- Auto-tuning approach

Thank you for your attention!

A. Heinecke, G. Henry, M. Hutchinson, H. Pabst, LIBXSMM: Accelerating small matrix multiplications by runtime code generation, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 84:1–84:11.