Fast Matrix-Vector Multiplication in Kronecker Form

Oguz Kaya and Olivier Coulaud

1 Background and Motivation

Tensors, or multi-dimensional arrays, are structures that generalize vectors and matrices to higher dimensions. Specifically, $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_D}$ is an D-dimensional tensor with elements $x_{i_1,\ldots,i_D} \in \mathbb{R}$ where $1 \leq i_d \leq N_d$ for all $1 \leq d \leq D$. Tensor-based computations have witnessed a remarkable increase in popularity in the last two decades due to great advancements in tensor theory and algorithms as well as their expressive power for high-dimensional algebraic structures that appear in fundamental applications in computational biology, chemistry, physics, signal processing, data analysis, and machine learning. Tensors are particularly useful when the problem at hand either has or can be reformulated to exhibit a high-dimensional structure. In this formulation, "high-dimensional" matrices and vectors of size $N^D \times N^D$ and of size N^D, respectively, are expressed as $2D$- and D-dimensional tensors, respectively, of size N in each dimension. In most cases, the inherent structure and properties of the problem that form these matrices/vectors (and corresponding tensors) imply a "low-rank" property; matrices and vectors can be expressed using polynomial number of elements in N and D (instead of N^{2D} or N^D elements), which is rendered possible via so-called *tensor decompositions*. Once matrices and vectors are expressed in this form, all matrix and vector operations such as matrix-vector multiplication and basic vector arithmetic (addition, subtraction, multiplication/division by a scalar, inner product) are performed under this "compressed" scheme with tremendous gains in terms of computational and memory costs.

One of the most popular tensor decompositions used in this context is called *tensor-train decomposition*. A matrix of size $N^D \times N^D$ is expressed via *tensor-train network* of D 4-dimensional tensors of size $N \times N \times R_A \times Q_A$, where (R_A, Q_A) denotes the rank of the matrix in the tensor-train form. Similarly, a vector $\mathbf{x}$ of size N^D comprises D 3-dimensional tensors of size $N \times R_x \times Q_x$. An anologous way to interpret these tensors would be to consider them as matrices (or vectors) of size $N \times N$ (or N), where each element in the matrix (vector) is a matrix of size $R_A \times Q_A$. We will be exclusively working on a fundamental kernel involving the multiplication of such *matrix/vector of matrices* described in what follows.

2 Matrix-vector multiplication in Kronecker form

Matrix-vector multiplication is a fundamental computational kernel in scientific computing whose optimization plays a key role in obtaining high performance linear and non-linear solvers. For a given matrix $\mathbf{A} \in \mathbb{R}^{M \times N}$ and a vector $\mathbf{x} \in \mathbb{R}^N$, the multiplication $\mathbf{y} = \mathbf{A}\mathbf{x}$ is defined as $y_i = \sum_{j=1}^{N} A_{i,j} x_j$ where $\mathbf{y} \in \mathbb{R}^M$.

Here, we will consider a generalized version of the standard matrix-vector multiplication, involving a *matrix of matrices* and a *vector of matrices*, i.e., $\mathbf{A}_{i,j}, \mathbf{x}_j$ and $\mathbf{y}_i$ are matrices. In describing this computation, we are in need of a special operation called *Kronecker product* on matrices, which is defined in what follows.

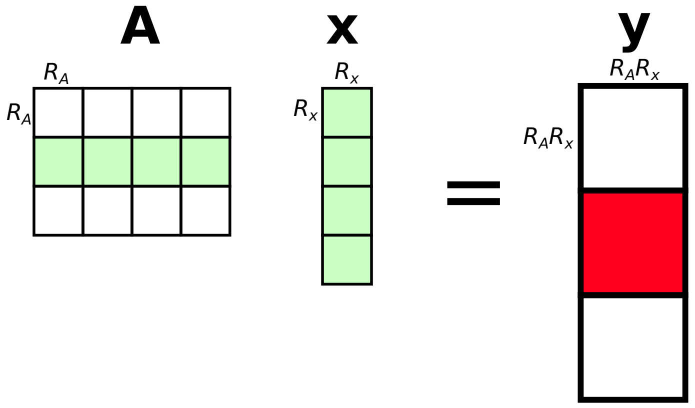
The *Kronecker product* of two matrices $\mathbf{B} \in \mathbb{R}^{M_B \times N_B}$ and $\mathbf{C} \in \mathbb{R}^{M_C \times N_C}$ is denoted by the operation $\otimes$ and results in a matrix $\mathbf{D} = \mathbf{B} \otimes \mathbf{C}, \mathbf{D} \in \mathbb{R}^{M_B M_C \times N_B N_C}$, which is defined as

$$\mathbf{D} = \mathbf{B} \otimes \mathbf{C} = \begin{pmatrix} B_{1,1}\mathbf{C} & \cdots & B_{1,N_B}\mathbf{C} \\ \vdots & \ddots & \vdots \\ B_{M_B,1}\mathbf{C} & \cdots & B_{M_B,N_B}\mathbf{C} \end{pmatrix} \tag{1}$$

We now define an operation we call *matrix-vector multiplication in Kronecker form (MxVK)* as follows. Given a matrix of $M \times N$ matrices $\mathbf{A}$ such that $\mathbf{A}_{i,j} \in \mathbb{R}^{R_A \times Q_A}$ for some fixed R_A, Q_A for all $1 \leq i, j \leq N$, and a vector of N matrices $\mathbf{x}$ such that $\mathbf{x}_j \in \mathbb{R}^{R_x \times Q_x}$ for some fixed R_x, Q_x for all $1 \leq j \leq N$, the result of the MxVK yields a vector $\mathbf{y} = \mathbf{A}\mathbf{x}$ of M matrices with entries

$$\mathbf{y}_i = \sum_{j=1}^{N} \mathbf{A}_{i,j} \otimes \mathbf{x}_j, \quad \mathbf{y}_i \in \mathbb{R}^{R_A R_x \times Q_A Q_x}$$

Note that when $R_A = Q_A = R_x = Q_x = 1$, MxVK reduces to standard matrix-vector multiplication as $\mathbf{A}_{i,j}$,$\mathbf{x}_j$ and $\mathbf{y}_i$ become scalars and Kronecker product reduces to scalar multiplication. We provide a pictorial representation of this operation in the following figure in which the computation of $\mathbf{y}_2 = \sum_{j=1}^{N} \mathbf{A}_{2,j} \otimes \mathbf{x}_j$ is highlighted.



A D-dimensional tensor/vector in tensor-train form comprises D matrices/vectors $\mathbf{A}^{(d)}/\mathbf{x}^{(d)}$, $0 \leq d < D$ in this form each potentially having different sizes $M_A^{(d)}, N_A^{(d)}$. Their ranks, however, are linked such that $Q_A^{(d-1)} = R_A^{(d)}$. Moreover, for the first and the last matrices, we always have $R_A^{(0)} = Q_A^{(D-1)} = 1$. Therefore, we rather denote these ranks in an array $R_A(0), R_A(1), \ldots, R_A(D)$ ($R_A(0) = R_A(D) = 1$), in which case the matrix in the dimension d obtains the ranks $R_A(d)$ and $R_A(d+1)$, respectively. Matrix-vector multiplication in this form involves an MxVK in each dimension to obtain $\mathbf{y}^{(d)} = \mathbf{A}^{(d)}\mathbf{x}^{(d)}$ (and there it is your first potential parallelism!).

Carrying out matrix-vector multiplication in tensor-train form is a fundamental kernel in low-rank tensor computations and constitutes one of the most expensive steps in the context of an iterative solver. The goal in this task is to implement very efficient matrix-vector multiplication kernels and effectively parallelize them with different paradigms using OpenMP.

You are already given a skeleton code `ttmatvec.cpp` that loads a matrix and a vector in tensor train form (`TTMat` and `TTVec`), multiplies them, then writes the results back into another file. It involves reference non-optimized sequential implementation provided in `ttvec.h/.cpp` and `ttmat.h/.cpp`. You will be implementing your own `ttvec.h/.cpp` and `ttmat.h/.cpp` that provides `TTMat` and `TTVec` structures/classes with all functions whose signatures are provided in `ttmatvec.cpp` so that we can replace the reference implementation with your code to test its performance.

In order to create `TTMat`s and `TTVec`s, we provide two executables `create-ttmat` and `create-ttvec` that generate random structures of specified sizes in a binary file. Run these executables without parameters to see their usage. Note that in these executables, to create a D dimensional TTMat/TTVec, we pass $D-1$ ranks as the rank in the first and the last dimension are always 1.

The first entry in the binary file for a `TTMat` is an integer representing D. This is followed by D integers representing the number of rows of matrices in each dimension (i.e. $M_A^{(d)}$). The next D integers provide the number of columns of matrices in each dimension (i.e., $N_A^{(d)}$. Afterwards, there are $D-1$ integers containing the ranks $R_A(1), \ldots, R_A(D-1)$ ($R_A(0)$ and $R_A(D)$ are always 1 hence not given in the file). In the rest of the file, we have `double` elements representing the matrices in dimensions $0, \ldots, D-1$, respectively. The matrix for the dimension d contains $M_A^{(d)} \times N_A^{(d)} \times R_A(d) \times R_A(d+1)$ `double` entries. Each matrix of size $M_A^{(d)} \times N_A^{(d)}$ in dimension d is stored in column-major layout, and each block of size $R_A^{(d)} \times R_A^{(d+1)}$ within this matrix is also stored in column-major layout. `TTVec` binary file is defined similarly, except that there is no $N_A^{(d)}$. Have a look at the load/save/print routines in `ttmat.cpp` and `ttvec.cpp` if you have any doubts about the data orientation. In your implementation, you should provide an input/output interface (with functions `loadTTMat/loadTTVec/saveTTVec` that can read/write a TT structure in a binary file. In your internal data structures for the matrix-vector multiplication algorithm, you are free to change the data orientation and storage as you will, but the resulting vector that you write to output must follow this column-major layout so that we can compare the results. Setup your internal data structures in `loadTTMat` and `loadTTVec`. These two function calls will not be timed for performance, so you can fine-tune your layout here without worrying about timing. Any computation in these functions pertaining to actual matrix-vector multiplication will be considered cheating, however!

Once your implementation writes the resulting vector $\mathbf{y}$ in a binary file, you can compare your results with the results of the reference implementation using `compare-ttvec` executable. Make sure your implementation is CORRECT before being OPTIMIZED, as you will get 0 score for a test case if the result is incorrect.

Do not hesitate to take your time to read the provided code and have a look at matrix/vector data structures. It is always insightful to read the reference implementation and understand how it works, before proceeding to your own implementation. You will see in the code that MxVK function utilizes another subroutine to perform the Kronecker product of two matrices as in (1), then performs the MxVK using this subroutine as in (2).

3 Tasks

3.1 Optimizing sequential MxVK

In this first task, your goal is to optimize the given MxVK kernel as much as possible. Take particularly into consideration the fact that matrices $\mathbf{A}_{i,j}$ and $\mathbf{X}_j$ are relatively small (e.g. ≤ 128 in each dimension). A good start would be determining the total number of flops performed in MxVK, the total amount of data "touched" during this computation (or total mops (memory operations)) in terms of N, R_A , and R_x. Finally, $\#flops/\#mops$ gives the computational intensity of this operation, which you should aim to attain with an optimized implementation. Once you are able to attain the computational intensity, do not hesitate to employ vectorization using AVX to crunch all this intensity if possible.

Explain each optimization technique you employed, and report the time difference with respect to previous baseline. Also report how well your code performs with respect to different set of parameters (i.e., small or large $M_A, N_A, R_A, Q_A, M_x, R_x, Q_x$), and try to fine-tune your code for edge cases if possible.

You are **NOT** allowed to use any sort of parallelization for this task. You are indeed allowed, however, to employ this optimized kernel in the following tasks requiring parallelization.

3.2 Parallelization using OpenMP

Now that you have a working sequential implementation, it is time to optimize it to the fullest and obtain the best performance. Here, you will parallelize the optimized sequential implementation you provided in the prevoius case using OpenMP parallel constructs. You are free to use parallel loops or sections. You are also welcome to make significant changes to your sequential implementation, in which case you would need to report why you needed such a change from a performance point of view. Make sure your paralellization scales acceptably well for pathological input sizes as well (i.e., N_A very small, R_A small, R_x large, etc.). Explain your parallelization strategy, how you divide the work among threads, achieve load balance, handle data dependencies, perform synchronization (if necessary), etc. Think about optimizing your parallelization on a single as well as multiple NUMA sockets as we will test the performance using multiple sockets as well. To restrain your code to work on a single socket, use the command $numactl - N0./ttmatvec......$

3.3 Parallelization using OpenMP tasks

You are **NOT** allowed to have any other means of parallelization for this assignment except OpenMP tasks.

You will now do another parallel implementation, but this time using OpenMP tasks. Note that your implementation should **NOT** be a simple re-wrapping of your previous parallelization with loops; you should try to extract finer-grain parallel tasks that have elegant output independence hence can be executed in parallel efficiently, but are coarse enough that it would not kill the performance due to tasking overhead. The performance will be similarly tested on a single and multiple NUMA sockets.

4 General guidelines and considerations

- You can assume that $1 \leq R_A, R_x \leq 128$, and $1 \leq N \leq 1024$.
- It might be useful to take into consideration the machine parameters (particularly the cache size at different levels, which you can query using `lscpu` command on Linux).
- Grading will be based on the speed of execution of your kernels. We will run each kernel multiple times and take the median. You should also be writing a report summarizing your approaches for optimization, and performance gains you obtained from each approach.
- The quality of your code (documentation, indentation, using proper functions, etc.) will also have an impact on your final score (i.e. we will subtract some points depending on how ugly the code is .)).

- Since we focus on parallelization, you are also encouraged to parallelize the work among the group members. For instance, sequential optimization and parallelization could be done in parallel! And task-based parallelization and OpenMP parallelization are independent as well!.