TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Parallel Optimization Algorithms for Very Large Tensor Decompositions

by

Ioannis Marios Papagiannakos

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DIPLOMA DEGREE OF

ELECTRICAL AND COMPUTER ENGINEERING

October 2019

THESIS COMMITTEE

Professor Athanasios P. Liavas, *Thesis Supervisor*
Professor George N. Karystinos
Associate Professor Vasilis Samoladas

Abstract

Tensors are generalizations of matrices to higher dimensions and are very powerful tools that can model a wide variety of multi–way data dependencies. As a result, tensor decompositions can extract useful information out of multi–aspect data tensors and have witnessed increasing popularity in various fields, such as data mining, social network analysis, biomedical applications, machine learning etc. Many decompositions have been proposed, but in this thesis we focus on Tensor Rank Decomposition or Canonical Polyadic Decomposition (CPD) using Alternating Least Squares (ALS). The main goal of the CPD is to decompose tensors into a sum of rank–1 terms, a procedure more difficult than its matrix counterpart, especially for large-scale tensors. CP decomposition via ALS consists of computationally expensive operations which cause performance bottlenecks. In order to accelerate this method and overcome these obstacles, we developed two parallel versions of the ALS that implement the CPD. The first one uses the full tensor and runs in parallel on heterogeneous & shared memory systems (CPUs and GPUs). The second one decomposes the tensor in parallel using small random block samples and runs on homogeneous & shared memory systems (CPUs).

Acknowledgements

First of all, I would like to thank my thesis supervisor, Professor Athanasios Liavas, for his continuous guidance throughout this work. Also, my friends and colleagues P. Karakasis, N. Siaminou and C. Kolomvakis for all the help they provided during my thesis. Finally, I would like to thank my family and my close friends for their support and encouragement throughout my study years.

Table of Contents

List of Figures

List of Abbreviations

ALS	Alternating Least Squares
API	Application Programming Interface
BCD	Block Coordinate Descent
CANDECOMP	Canonical Decomposition
CPD	Canonical Polyadic Decomposition
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
MIMD	Multiple Instruction - Multiple Data
MTTKRP	Matricized Tensor Times Khatri-Rao Product
OpenMP	Open Multi-Processing
PARAFAC	Parallel Factor Analysis
RBS	Randomized Block Sampling
SGD	Stochastic Gradient Descend
SIMT	Single Instruction - Multiple Threads

Chapter 1

Introduction

1.1 Problem Description (Tensor factorization)

Tensors are mathematical structures that can be described as multidimensional arrays of numerical values and, therefore, generalize matrices to multiple dimensions. Tensors and tensor decompositions are important tools that can model multi-way data dependencies. Tensor decomposition (or factorization) can extract useful information, which would otherwise be lost when analysing the data by matrix factorization approaches by collapsing some of the modes. Tensor decomposition techniques started in the first quarter of the 20^{th} century in applications related to psychometrics, but gained great popularity recently, in a variety of fields, such as neuroscience, data mining, machine learning [1],[2],[3].

There are various tensor factorization techniques, but we will focus on one of the most popular, known as Canonical Polyadic Decomposition (CPD), also referred as CANDECOMP or PARAFAC (Parallel Factor Analysis). CPD is a rank decomposition technique and the main goal is to express a given tensor as the sum of a finite number of rank-one tensors. In order to solve this problem, we use the *ALS* algorithm which will be presented later.

1.2 Definitions and Notation

Vectors are denoted by lower case bold letters (e.g. $\mathbf{x}$), matrices by capital bold letters (e.g. $\mathbf{X}$), and tensors by calligraphic upper case bold letters (e.g. $\boldsymbol{\mathcal{X}}$). Elements of either vectors, or matrices, or tensors are denoted by non bold letters, and the appropriate set of indices. For example for a matrix $\mathbf{A}$, element in the i^{th} row and j^{th} column is denoted either as a_{ij} or as $A_{(i,j)}$.

Definition 1 *The* ***outer product*** *of two vectors* $\mathbf{a} \in \mathbb{R}^I$ *and* $\mathbf{b} \in \mathbb{R}^J$ *is denoted as* $\mathbf{a} \circ \mathbf{b} \in \mathbb{R}^{I\times J}$ *and gives a rank-one matrix. Likewise, a* ***3−way outer product*** *of any three vectors,* $\mathbf{a} \in \mathbb{R}^I$, $\mathbf{b} \in \mathbb{R}^J$, $\mathbf{c} \in \mathbb{R}^K$ *is denoted as* $\mathbf{a} \circ \mathbf{b} \circ \mathbf{c} \in \mathbb{R}^{I\times J\times K}$ *and gives a rank–one tensor with elements* $(\mathbf{a} \circ \mathbf{b} \circ \mathbf{c})(i,j,k) = a(i)b(j)c(k)$.

Definition 2 *The* ***Kronecker product*** *of matrices* $\mathbf{A} \in \mathbb{R}^{I\times R}$ *and* $\mathbf{B} \in \mathbb{R}^{J\times P}$ *is denoted as* $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{IJ\times RP}$ [4], *and is computed as follows*

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} \mathbf{A}_{(1,1)}\mathbf{B} & \dots & \mathbf{A}_{(1,J)}\mathbf{B} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{(I,1)}\mathbf{B} & \dots & \mathbf{A}_{(I,J)}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{IJ\times RP}. \tag{1.1}$$

Definition 3 *The* ***Khatri-Rao (or column-wise Kronecker) product*** *of matrices*

$\mathbf{A} \in \mathbb{R}^{I \times R}$ *and* $\mathbf{B} \in \mathbb{R}^{J \times R}$ *is denoted as* $\mathbf{A} \odot \mathbf{B} \in \mathbb{R}^{IJ \times R}$. *The Khatri-Rao product is computed as*

$$\mathbf{A} \odot \mathbf{B} = \left[\mathbf{A}_{(:,1)} \otimes \mathbf{B}_{(:,1)} \dots \mathbf{A}_{(:,R)} \otimes \mathbf{B}_{(:,R)} \right] \in \mathbb{R}^{IJ \times R}. \tag{1.2}$$

Definition 4 *The* ***Hadamard (or element-wise) product*** *of matrices* $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{I \times R}$ *is denoted as* $\mathbf{A} \circledast \mathbf{B} \in \mathbb{R}^{I \times R}$. *The Hadamard product of an element in* i^{th} *row and* j^{th} *column is computed as*

$$[\mathbf{A} \circledast \mathbf{B}]_{(i,j)} = \mathbf{A}_{(i,j)} \cdot \mathbf{B}_{(i,j)} \tag{1.3}$$

Definition 5 *The* ***Frobenius Norm*** *of a tensor* $\mathcal{X} \in \mathbb{R}^{I \times J \times \times K}$ *is defined as*

$$||\mathcal{X}||_F = \sqrt{\sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} \mathcal{X}(i,j,k)^2} \quad . \tag{1.4}$$

Definition 6 *The* ***order*** *of a tensor is the number of dimensions that it has. More precisely, scalars can be described as zeroth-order tensors, vectors as first-order tensors, matrices as second-order tensors, and any tensor having order n>2 (e.g.* $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$*) will be referred as nth-order tensor.* In this thesis, we focus on third-order tensors (Figure 1.1).

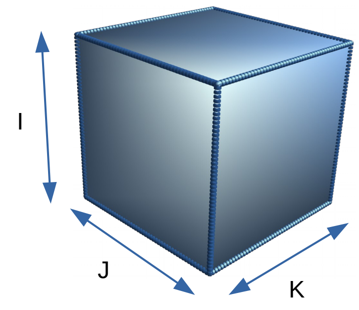

Figure 1.1: A third-order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$

Definition 7 *The* ***rank*** *of a tensor* $\mathcal{X}$ *is denoted as rank(*$\mathcal{X}$*) and defines the minimum number of rank-one tensors which are needed to produce* $\mathcal{X}$ *as their sum. For example, let* $\mathcal{X}$ *be a third-order tensor with rank(*$\mathcal{X}$*)* $= R$*, then*

$$\mathcal{X} = \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r. \tag{1.5}$$

Definition 8 *In general, we can extract lower order tensors from a nth-order tensor. In our case, from a third-order tensor, we can extract a first and second order one (vectors and matrices correspondingly). More precisely, if we fix all but one indices, a* ***fiber*** *is created, otherwise if we fix all but two indices, we create a* ***slice****. From a third-order tensor* $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$*, fibers are given as* $\mathbf{x}_{:jk}$*,* $\mathbf{x}_{i:k}$ *and* $\mathbf{x}_{ij:}$*, and slices are given as* $\mathbf{X}_{::k}$*,* $\mathbf{X}_{:j:}$ *and* $\mathbf{X}_{i::}$*.*

Definition 9 *The **Mode-n Matricization** of $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ is denoted as*

$$\mathbf{X}_{(1)} \in \mathbb{R}^{I\times JK}, \mathbf{X}_{(2)} \in \mathbb{R}^{J\times IK}, \mathbf{X}_{(3)} \in \mathbb{R}^{K\times IJ}, \tag{1.6}$$

and defines the operation that reorders a tensor into a matrix, by turning the mode-n fibers of tensor $\mathcal{X}$ into the columns of matrix $\mathbf{X}_{(n)}$.

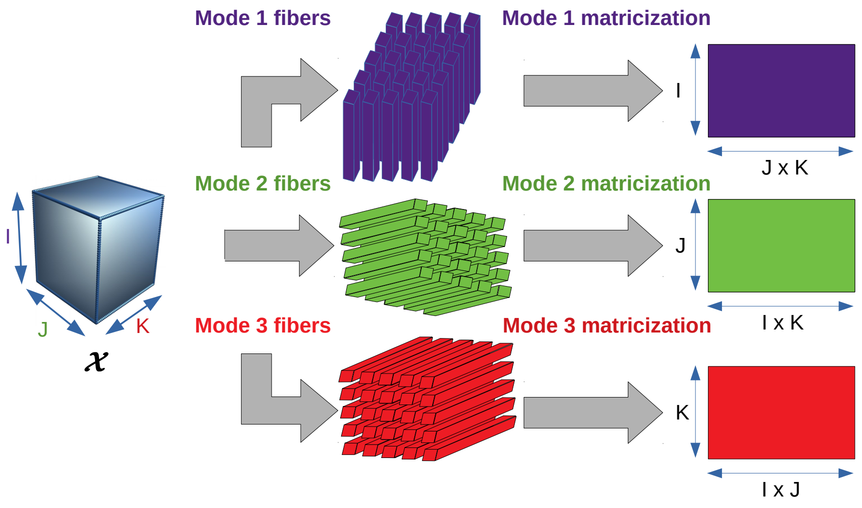

Figure 1.2: Mode-n Matricization of $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$

Definition 10 *The **Moore–Penrose pesudoinverse** of matrix $\mathbf{A} \in \mathbb{R}^{M\times N}$ is denoted as $\mathbf{A}^{\dagger}$* , is in $\mathbb{R}^{N\times M}$ and has the following properties

- $\mathbf{A}\mathbf{A}^{\dagger}\mathbf{A} = \mathbf{A}$
- $\mathbf{A}^{\dagger}\mathbf{A}\mathbf{A}^{\dagger} = \mathbf{A}^{\dagger}$
- $(\mathbf{A}\mathbf{A}^{\dagger})^T = \mathbf{A}\mathbf{A}^{\dagger}$
- $(\mathbf{A}^{\dagger}\mathbf{A})^T = \mathbf{A}^{\dagger}\mathbf{A}$

If $\mathbf{A} \in \mathbb{R}^{M\times M}$ is invertible, then

$$\mathbf{A}^{\dagger} = \mathbf{A}^{-1} \tag{1.7}$$

1.3 Structure

In Chapter 2, we present the ALS algorithm for tensor factorization and RBS; a randomized block sampling approach to CPD. In Chapter 3, we introduce OpenMP, an API designed for shared-memory parallelism and we present parallel implementations on CPD and RBS–CPD via ALS. At the end of this chapter, we also present the corresponding speedups obtained from the parallel versions. In Chapter 4, we present CUDA, a parallel computing platform and API for general computing on graphical processing units (GPUs), in which we implemented parts of ALS, in order to accelerate its execution. This chapter contains also the respective speedups obtained from the parallel version. Finally, in Chapter 5, we end our thesis with some ideas for improvement as a future work.

Chapter 2

Tensor Factorization

As mentioned before, CPD is one of the most popular tensor rank factorizations. The CPD of a three-mode tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, where rank($\mathcal{X}$)= R, is the sum of three-way outer products,

$$\mathcal{X} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \tag{2.1}$$

where, $\mathbf{a}_r \in \mathbb{R}^I$, $\mathbf{b}_r \in \mathbb{R}^J$, $\mathbf{c}_r \in \mathbb{R}^K$.
Using vectors $\mathbf{a}_r$, $\mathbf{b}_r$, $\mathbf{c}_r$ we construct the respective factor matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, which are formed as:

$$\begin{aligned} \mathbf{A} &= [\mathbf{a}_1 \quad \mathbf{a}_2 \quad ... \quad \mathbf{a}_R] \in \mathbb{R}^{I \times R}, \\ \mathbf{B} &= [\mathbf{b}_1 \quad \mathbf{b}_2 \quad ... \quad \mathbf{b}_R] \in \mathbb{R}^{J \times R}, \\ \mathbf{C} &= [\mathbf{c}_1 \quad \mathbf{c}_2 \quad ... \quad \mathbf{c}_R] \in \mathbb{R}^{K \times R}. \end{aligned}$$

Thus, equation 2.1 can also be written as:

$$\mathcal{X} = [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!] \tag{2.2}$$

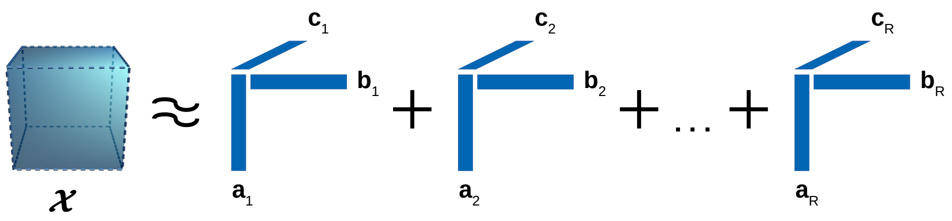

Figure 2.1: CP Decomposition

2.1 CPD using ALS

Alternating Least Squares (ALS) is a widely used algorithm for CPD. For a third-order tensor $\mathcal{X}$ with given rank R, in order to be decomposed, we solve the following optimization problem:

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} \left\| \mathcal{X} - \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \right\|_F^2 = \min_{\mathbf{A},\mathbf{B},\mathbf{C}} \left\| \mathcal{X} - [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!] \right\|_F^2 \tag{2.3}$$

Let $f_{\mathcal{X}}$ be the function used as quality measure for the above factorization problem. Function $f_{\mathcal{X}}$ is non–convex if we try to optimize all three factor matrices at once. However, if we fix two of them and try to optimize the non–fixed one, it reduces to a matrix least squares; and therefore it becomes convex. ALS is a type of block coordinate descent algorithm (BCD). This means that instead of using the whole gradient, we select a block of coordinates in each iteration. Cost function $f_{\mathcal{X}}$ can be expressed as:

$$\begin{aligned} f_{\mathcal{X}}(\mathbf{A},\mathbf{B},\mathbf{C}) &= \frac{1}{2}\left\|\mathbf{X}_{(1)} - \mathbf{A}(\mathbf{B}\odot\mathbf{C})^T\right\|_F^2 \\ &= \frac{1}{2}\left\|\mathbf{X}_{(2)} - \mathbf{B}(\mathbf{C}\odot\mathbf{A})^T\right\|_F^2 \\ &= \frac{1}{2}\left\|\mathbf{X}_{(3)} - \mathbf{C}(\mathbf{B}\odot\mathbf{A})^T\right\|_F^2 . \end{aligned} \tag{2.4}$$

ALS algorithm works repeatedly, until a terminating criterion is satisfied (e.g. convergence of cost function). The main steps are:

$$\begin{aligned} \mathbf{A}_{k+1} &\leftarrow \arg\min_{\mathbf{A}_k} f_{\mathcal{X}}(\mathbf{A}_k,\mathbf{B}_k,\mathbf{C}_k) \\ \mathbf{B}_{k+1} &\leftarrow \arg\min_{\mathbf{B}_k} f_{\mathcal{X}}(\mathbf{A}_k,\mathbf{B}_k,\mathbf{C}_k) \\ \mathbf{C}_{k+1} &\leftarrow \arg\min_{\mathbf{C}_k} f_{\mathcal{X}}(\mathbf{A}_k,\mathbf{B}_k,\mathbf{C}_k) \end{aligned}$$

The optimal solution of the above optimization problems is given by the following closed form:

$$\begin{aligned} \mathbf{A}^*_{k+1} &= \mathbf{X}_{(1)}[(\mathbf{B}_k\odot\mathbf{C}_k)]^\dagger = \mathbf{X}_{(1)}(\mathbf{B}_k\odot\mathbf{C}_k)(\mathbf{B}_k^T\mathbf{B}_k \circledast \mathbf{C}_k^T\mathbf{C}_k)^\dagger \\ &\overset{(1.7)}{=} \mathbf{X}_{(1)}(\mathbf{B}_k\odot\mathbf{C}_k)(\mathbf{B}_k^T\mathbf{B}_k \circledast \mathbf{C}_k^T\mathbf{C}_k)^{-1} \\ \mathbf{B}^*_{k+1} &= \mathbf{X}_{(2)}[(\mathbf{C}_k\odot\mathbf{A}_k)]^\dagger = \mathbf{X}_{(2)}(\mathbf{C}_k\odot\mathbf{A}_k)(\mathbf{C}_k^T\mathbf{C}_k \circledast \mathbf{A}_k^T\mathbf{A}_k)^\dagger \\ &\overset{(1.7)}{=} \mathbf{X}_{(2)}(\mathbf{C}_k\odot\mathbf{A}_k)(\mathbf{C}_k^T\mathbf{C}_k \circledast \mathbf{A}_k^T\mathbf{A}_k)^{-1} \\ \mathbf{C}^*_{k+1} &= \mathbf{X}_{(3)}[(\mathbf{B}_k\odot\mathbf{A}_k)]^\dagger = \mathbf{X}_{(3)}(\mathbf{B}_k\odot\mathbf{A}_k)(\mathbf{B}_k^T\mathbf{B}_k \circledast \mathbf{A}_k^T\mathbf{A}_k)^\dagger \\ &\overset{(1.7)}{=} \mathbf{X}_{(3)}(\mathbf{B}_k\odot\mathbf{A}_k)(\mathbf{B}_k^T\mathbf{B}_k \circledast \mathbf{A}_k^T\mathbf{A}_k)^{-1} \end{aligned}$$

Products $\mathbf{X}_{(1)}(\mathbf{B}\odot\mathbf{C})$, $\mathbf{X}_{(2)}(\mathbf{C}\odot\mathbf{A})$, $\mathbf{X}_{(3)}(\mathbf{B}\odot\mathbf{A})$ are referred as *Matricized Tensor Times Khatri–Rao Product (MTTKRP)*; and are of paramount importance since are a bottleneck operation in ALS.

Using the Khatri–Rao expression, the Matricized Tensor can be expressed approximately as:

$$\begin{aligned}\mathbf{X}_{(1)} &= \mathbf{A}(\mathbf{B} \odot \mathbf{C})^T \\ \mathbf{X}_{(2)} &= \mathbf{B}(\mathbf{C} \odot \mathbf{A})^T \\ \mathbf{X}_{(2)} &= \mathbf{C}(\mathbf{B} \odot \mathbf{A})^T\end{aligned} \tag{2.5}$$

Except of the aforementioned steps, ALS also contains two optional steps. The first one is the column normalization of each matrix factor and the second one is "acceleration"step. The first one offers stability to the algorithm [2]. Function *Normalize()* normalizes each column of updated factors $\mathbf{B}_{k+1}, \mathbf{C}_{k+1}$ to unit Euclidean norm, collecting all the power on the corresponding columns of $\mathbf{A}_{k+1}$. This function's output is denoted as $(\mathbf{A}^{\mathcal{N}}_{k+1}, \mathbf{B}^{\mathcal{N}}_{k+1}, \mathbf{C}^{\mathcal{N}}_{k+1})$. Function *Accelerate()* acts as an accelerating technique and despite the fact that this step has a significant computational cost, it can reduce the number of iterations that are needed for ALS to converge. It is a line search technique, similar to technique used in[5] and can be described shortly as follows.

After normalization step, we compute

$$\mathbf{A}_{new} = \mathbf{A}^{\mathcal{N}}_{k} + s_{k+1}(\mathbf{A}^{\mathcal{N}}_{k+1} - \mathbf{A}^{\mathcal{N}}_{k}), \qquad \mathbf{B}_{new} = \mathbf{B}^{\mathcal{N}}_{k} + s_{k+1}(\mathbf{B}^{\mathcal{N}}_{k+1} - \mathbf{B}^{\mathcal{N}}_{k}),$$
$$\mathbf{C}_{new} = \mathbf{C}^{\mathcal{N}}_{k} + s_{k+1}(\mathbf{C}^{\mathcal{N}}_{k+1} - \mathbf{C}^{\mathcal{N}}_{k}),$$

where s_{k+1} a decreasing positive number, computed as $s_{k+1} = (k+1)^{\frac{1}{3}}$.

If $f_{\mathcal{X}}(\mathbf{A}_{new}, \mathbf{B}_{new}, \mathbf{C}_{new}) \leq f_{\mathcal{X}}(\mathbf{A}_{k+1}, \mathbf{B}_{k+1}, \mathbf{C}_{k+1})$ then the acceleration step is successful and we set each factor as: $\mathbf{A}_{k+1} = \mathbf{A}_{new}$, $\mathbf{B}_{k+1} = \mathbf{B}_{new}$, $\mathbf{C}_{k+1} = \mathbf{C}_{new}$, else it is ignored and we keep the normalized factors, $\mathbf{A}_{k+1} = \mathbf{A}^{\mathcal{N}}_{k+1}$, $\mathbf{B}_{k+1} = \mathbf{B}^{\mathcal{N}}_{k+1}$, $\mathbf{C}_{k+1} = \mathbf{C}^{\mathcal{N}}_{k+1}$.

Presenting more theoretical analysis about this function is out of the scope of this thesis. For further reading on accelerating techniques on CPD, one can read [6].

It is important to mention that the way the algorithm is initialized has a big impact on performance. ALS may take several steps to converge and it is not always guaranteed that a global optimum is attained. Thus, in our thesis, the terminating criterion of ALS in not only determined by convergence, but also, by the maximum number of iterations. Algorithm 1 describes how ALS works.

We expect that the most time consuming steps are those where the factors are updated. Each step consists of one Khatri–Rao product, one Hadamard product, one matrix inversion and four matrix multiplications. The Khatri–Rao product of two factors, say $C \in \mathbb{R}^{K \times R}$ and $B \in \mathbb{R}^{K \times R}$, has computational complexity $\mathcal{O}(KJR)$ (KJR element–wise multiplications). The Hadamard product of two matrices, $C \in \mathbb{R}^{K \times R}$ and $B \in \mathbb{R}^{K \times R}$, has computational complexity $\mathcal{O}(KR)$ (KR element–wise multiplications). The linear system solution of a square matrix $A \in \mathbb{R}^{N \times N}$ has in the worst case scenario computational complexity $\mathcal{O}(N^3)$ (matrix inversion). Also, naive matrix multiplication of two matrices, say $A \in \mathbb{R}^{M \times N}$ and $B \in \mathbb{R}^{N \times K}$, has computational complexity $\mathcal{O}(MNK)$. Therefore, the

Algorithm 1 CPD-ALS algorithm

```
1: procedure CPD-ALS($\mathcal{X}$,$R$,$\epsilon$,$MAX\_ITERS$)
2:     initialize $k$, $f_{\mathcal{X}}$ and matrix factors $\mathbf{A}_0, \mathbf{B}_0, \mathbf{C}_0$
3:     while $f_{\mathcal{X}}/\|\mathcal{X}\|_F^2 > \epsilon$ or $k < MAX\_ITERS$ do
4:         $\mathbf{A}_{k+1} \leftarrow \mathbf{X}_{(1)}(\mathbf{B}_k \odot \mathbf{C}_k)(\mathbf{C}_k^T\mathbf{C}_k \circledast \mathbf{B}_k^T\mathbf{B}_k)^{-1}$
5:         $\mathbf{B}_{k+1} \leftarrow \mathbf{X}_{(2)}(\mathbf{C}_k \odot \mathbf{A}_{k+1})(\mathbf{C}_k^T\mathbf{C}_k \circledast \mathbf{A}_{k+1}^T\mathbf{A}_{k+1})^{-1}$
6:         $\mathbf{C}_{k+1} \leftarrow \mathbf{X}_{(3)}(\mathbf{B}_{k+1} \odot \mathbf{A}_{k+1})(\mathbf{B}_{k+1}^T\mathbf{B}_{k+1} \circledast \mathbf{A}_{k+1}^T\mathbf{A}_{k+1})^{-1}$
7:         Compute $f_{\mathcal{X}}/\|\mathcal{X}\|_F^2$
8:         $(\mathbf{A}_{k+1}^{\mathcal{N}}, \mathbf{B}_{k+1}^{\mathcal{N}}, \mathbf{C}_{k+1}^{\mathcal{N}}) \leftarrow$ Normalize$(\mathbf{A}_{k+1}, \mathbf{B}_{k+1}, \mathbf{C}_{k+1})$
9:         $(\mathbf{A}_{k+1}, \mathbf{B}_{k+1}, \mathbf{C}_{k+1}) \leftarrow$ Accelerate$(\mathbf{A}_{k+1}^{\mathcal{N}}, \mathbf{A}_k^{\mathcal{N}}, \mathbf{B}_{k+1}^{\mathcal{N}}, \mathbf{B}_k^{\mathcal{N}}, \mathbf{C}_{k+1}^{\mathcal{N}}, \mathbf{C}_k^{\mathcal{N}})$
10:        $k \leftarrow k + 1$
11:    end while
12:    return matrix factors $(\mathbf{A}_k, \mathbf{B}_k, \mathbf{C}_k)$
13: end procedure
```

computational complexity of each step is computed as :

$$\underbrace{\mathcal{O}(IJKR)}_{MTTKRP} + \underbrace{\mathcal{O}(KJR)}_{\text{Khatri–Rao}} + \underbrace{\mathcal{O}(KR^2)}_{C^TC} + \underbrace{\mathcal{O}(JR^2)}_{B^TB} + \underbrace{\mathcal{O}(R^3)}_{\text{lin. sys. sol. of Matrix}} + \underbrace{\mathcal{O}(R^2)}_{Hadamard} = \mathcal{O}\big(IJKR\big)$$

We ignore terms $\mathcal{O}(KR^2)$,$\mathcal{O}(JR^2)$, $\mathcal{O}(R^3)$, $\mathcal{O}(R^2)$ since typically for a large tensor $\mathcal{X}$, rank $R < min(I, J, K)$.

In order to validate the above theoretical analysis, we developed a serial version of ALS algorithm and we measured the execution time of each step. We observed that for a large enough tensor (e.g. $500 \times 500 \times 500$ and rank $R = 20$ on a 8GB memory system) the most time consuming steps were: update of each factor **A**, **B** and **C** ($\approx 25\%$ of total time each) and also acceleration step ($\approx 24\%$). These four steps consume almost 99% of the total time and cause the main bottleneck.

2.2 RBS–CPD using ALS

As mentioned in the previous section, ALS is a widely used block coordinate descent algorithm, used to compute CPD. However, this algorithm has big computational and memory complexity, especially for large–scale tensors. In order to reduce this problem, many methods have been proposed and one of them is randomized block sampling (RBS).

RBS–CPD is a combination of two optimization techniques, block coordinate descent and stochastic gradient descent (SGD) [7]. The first one was introduced previously and the second one we will presented shortly in this section.

SGD is a simple and also powerful technique, used mostly in convex optimization and also other applications e.g. machine learning [8]. Suppose that we want to minimize a decomposable function f with respect to $\mathbf{x}$

$$f(\mathbf{x}) = \frac{1}{N}\sum_{n=1}^{N} f_n(\mathbf{x})\,. \tag{2.6}$$

In order to update parameter $\mathbf{x}$ in each k^{th} iteration, gradient ∇f is estimated from a random sample point $n_k \in [1, N]$ instead of the whole vector $\mathbf{x}$. This can be generalized on CPD. More specifically, problem (2.3) can be written as an optimization problem of a decomposable function f

$$f = \frac{1}{2}\left\| \mathcal{X} - \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \right\|_F^2 = \frac{1}{2}\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K}\left(\mathcal{X}(i,j,k) - \sum_{r=1}^{R}\Big(a_r(i)b_r(j)c_r(k)\Big)\right)^2 . \tag{2.7}$$

Since f is decomposable, CPD can be solved using ALS, but instead of applying it on the full tensor and factors, we select randomly sampled blocks (RBS–CPD). Furthermore, in order to update the whole factors at the same rate, we use a sampling operator (modified RBS–CPD). Sampling operator ensures that blocks are not overlapping, allowing the blocks to be decomposed in parallel.

For each l^{th} sub–problem that will be solved in parallel, let $\mathcal{B}_n^l \subseteq \mathcal{I}_n$, be the subset that contains each block B_n of indices. Each subset $\mathcal{B}_n^l$ contains indices from index set $\mathcal{I}_n$ of the full tensor $\mathcal{X}$. We also define integer $Q_n = I_n/B_n$ as the number of blocks per dimension. Finally, let $\mathcal{X}_{sub}$ be the sampled sub–tensor and $\mathbf{A}_{sub}, \mathbf{B}_{sub}, \mathbf{C}_{sub}$ be each corresponding sampled sub–factor.

Factors are updated at each k^{th} iteration, where each one contains Q_n inner iterations. In order to ensure that each factor is fully updated in every iteration and that each block is mutually independent, sampling operator works as follows. During each inner iteration l, new blocks of indices and new samples are generated. From each factor, we sample B_n rows and from tensor $\mathcal{X}$ we sample blocks of size $B_1 \times B_2 \times B_3$. At the first iteration ($k = 1$), indices $\mathcal{B}_n^l$ are selected consecutively from $\mathcal{I}_n$ based to the following form $\mathcal{B}_n^l = I_n((l-1) \cdot B_n + 1 : l \cdot B_n)$, where $l \leq Q_n$, until inner iterations are ended. Then, at the next iteration, set $\mathcal{I}_n$ is shuffled and the same procedure is repeated, except that this time block indices are sorted in order to succeed better locality. This leads to performance improvement, especially for larger block size.

As a convergence measure we used *relative factor change function*, instead of $f_{\mathcal{X}}$ (2.4) which has much more computational complexity. Relative factor change function is computed as :

$$f_{rel}(\mathbf{A}_{k+1}, \mathbf{A}_k) = \frac{\|\mathbf{A}_{k+1} - \mathbf{A}_k\|_2}{\|\mathbf{A}_k\|_2}. \tag{2.8}$$

The algorithm is said to have converged if

$$max(f_{rel}(\mathbf{A}_{k+1}, \mathbf{A}_k), f_{rel}(\mathbf{B}_{k+1}, \mathbf{B}_k), f_{rel}(\mathbf{C}_{k+1}, \mathbf{C}_k)) < \epsilon, \tag{2.9}$$

where ϵ is a small constant, e.g. 10^{-2}.

Figure (2.2) illustrates how sampling operator works and Algorithm 2 summarizes RBS–CPD.

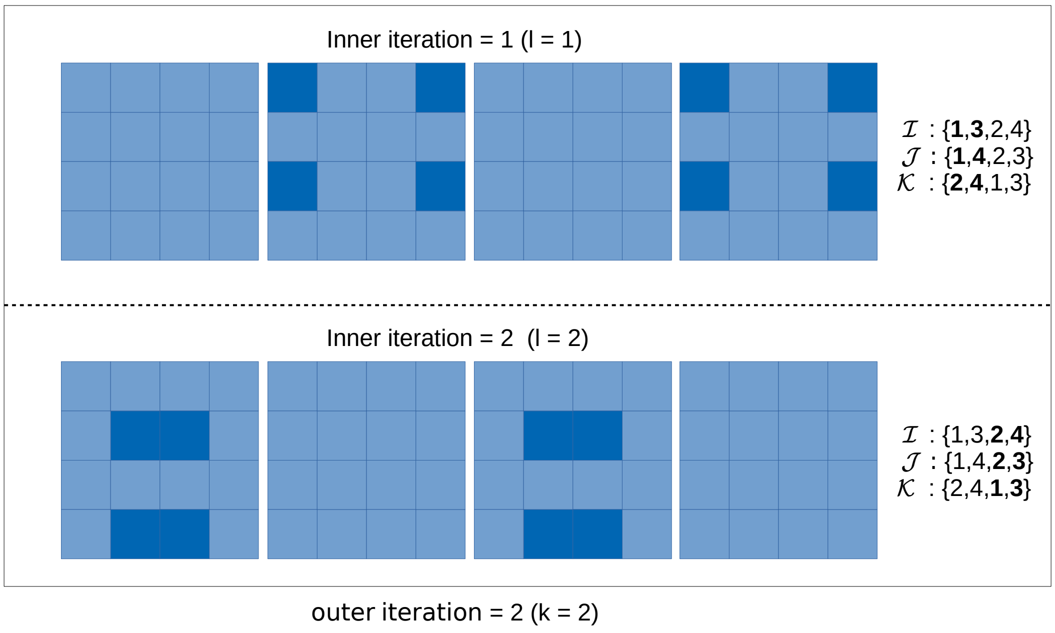
Inner iteration = 1 (l = 1)
$\mathcal{I}$: {**1**,**3**,2,4}
$\mathcal{J}$: {**1**,**4**,2,3}
$\mathcal{K}$: {**2**,**4**,1,3}
Inner iteration = 2 (l = 2)
$\mathcal{I}$: {1,3,**2**,**4**}
$\mathcal{J}$: {1,4,**2**,**3**}
$\mathcal{K}$: {2,4,**1**,**3**}
outer iteration = 2 (k = 2)

Figure 2.2: Illustration of block sampling operator for a third–order tensor $\mathcal{X} \in \mathbb{R}^{4\times4\times4}$ using a block size of $2 \times 2 \times 2$. We show how sampling is carried out on the frontal slices of tensor $\mathcal{X}$. Bold numbers indicate which indices are sampled at each inner iteration.

Algorithm 2 Modified RBS–CPD algorithm

```
 1: procedure RBS–CPD($\mathcal{X}$)
 2:     initialize $k$ and matrix factors $\mathbf{A}, \mathbf{B}, \mathbf{C}$
 3:     while no convergence do
 4:         $l = 1$
 5:         if $k > 1$ then
 6:             Shuffle indices $I_n$
 7:         end if
 8:         while $l \leq Q_n$ do
 9:             Generate $\mathcal{B}_n^l$ for $n = 1, 2, 3$
10:             Sample $\mathbf{A}_{sub}^{(l)}, \mathbf{B}_{sub}^{(l)}, \mathbf{C}_{sub}^{(l)}, \mathcal{X}_{sub}^{(l)}$
11:             new $\mathbf{A}_{sub}^{(l)} \leftarrow$ update$\left(\mathcal{X}_{sub(1)}^{(l)}, \mathbf{B}_{sub}^{(l)}, \mathbf{C}_{sub}^{(l)}\right)$
12:             new $\mathbf{B}_{sub}^{(l)} \leftarrow$ update$\left(\mathcal{X}_{sub(2)}^{(l)}, \mathbf{C}_{sub}^{(l)}, \text{new } \mathbf{A}_{sub}^{(l)}\right)$
13:             new $\mathbf{C}_{sub}^{(l)} \leftarrow$ update$\left(\mathcal{X}_{sub(3)}^{(l)}, \text{new } \mathbf{B}_{sub}^{(l)}, \text{new } \mathbf{A}_{sub}^{(l)}\right)$
14:             Merge new sub-factors with corresponding factors
15:             $l \leftarrow l + 1$
16:         end while
17:         $k \leftarrow k + 1$
18:     end while
19:     return matrix factors $\mathbf{A}, \mathbf{B}, \mathbf{C}$
20: end procedure
```

Chapter 3

Parallel Implementation using OpenMP

Both algorithms described in previous chapter, ALS and RBS, can be performed in parallel. For that reason, we will present OpenMP (MP stands for *multiprocessing*), an API developed for shared–memory parallel programming. The reason that we selected OpenMP rather than other interfaces, like POSIX threads (*Pthreads*), is that the first one is higher level, allowing us to parallelize tasks and assign them to threads easier.

3.1 Introduction to Parallel Computing

In parallel computing there are four different types of computer architectures according to *Flynn's Taxonomy.* Systems are classified according to the number of instruction streams and the number of data streams that can be managed simultaneously. Modern CPUs are classified as MIMD (*Multiple Instruction – Multiple Data*).

3.1.1 Multiple Instruction - Multiple Data

MIMD systems support multiple cores which operate asynchronously on multiple data streams. This means that each core executes independent instructions. MIMD systems can have either distributed memory (multi–node with distributed memory) or shared memory (multiprocessor with shared memory). In this thesis, we will focus on shared memory computers.

3.1.2 Shared-memory systems

Most shared-memory systems use one or more multi–core processors. Multiple cores that belong to the same chip are organized in groups, known as *sockets.* A typical desktop PC has a single socket, while standard servers use two to four sockets that share the same memory. Usually, each core has a private level 1 cache and each socket has a private level 2 cache, which is shared between cores. Each socket can be connected to a global main memory or to a local memory, which in both cases is shared between sockets. Systems that have the first topology are characterized as *uniform memory access* (UMA) and therefore, the memory access time between cores that belong to different sockets is equal. In contrast, the second ones are called *nonuniform memory access* (NUMA), which means that memory access time differs between cores that are directly connected to local memory and cores that belong to different sockets. This makes UMA systems easier to program, since the programmer does not need to worry about different access times for memory locations. On the contrary, NUMA systems have higher overall memory bandwidth and also have the potential to use larger amount of memory.

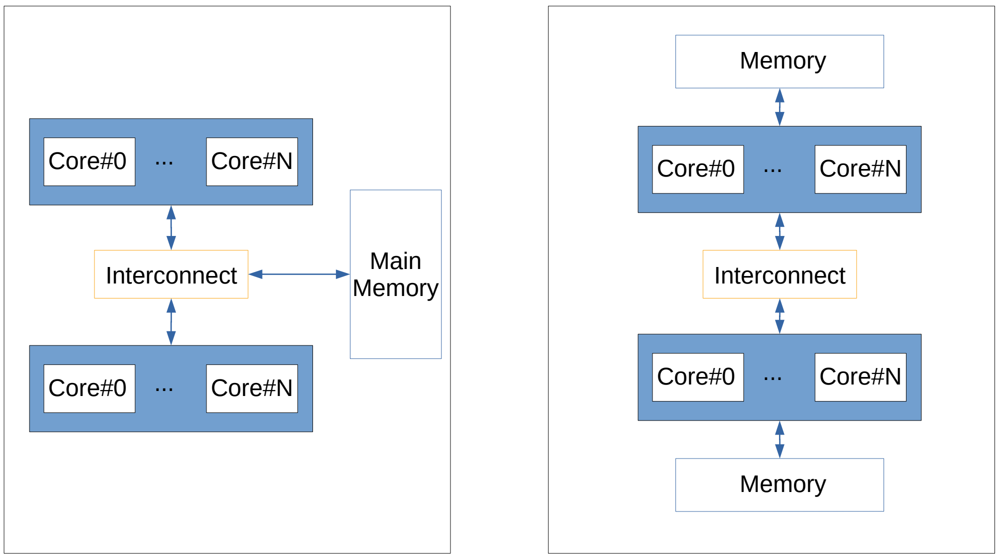

Figure 3.1: Illustration of UMA (left) and NUMA (right) architecture.

3.2 OpenMP

Before talking about OpenMP, we should define the terms process and thread. Process is an instance of a program that is being executed on a processor. Thread is a basic unit of CPU utilization and one ore more threads are contained within processes, so they can use the same executable and they usually share the same resources (except stack and program counter). Processes, in contrast, can only share resources through techniques such as shared memory and message passing. In most systems, context switch time between threads is lower compared to process context switch. This is because threads are "lightweight" processes. Therefore, multithreading is a better parallelism scheme than multiprocessing, in a shared–memory system.

OpenMP is directive–based programming model designed as an extension of C and C++ [9] [10]. This can be done using special preprocessor instructions, known as *pragmas*, which are added in order to extend existing capabilities of basic C/C++ specification. More precisely, pragmas in OpenMP always begin with `#pragma omp`. OpenMP consist of a collection of directives and also a library of functions and macros which are included in header file `<omp.h>`.

OpenMP directives

The most basic directive is `parallel` and it specifies that the block of code that follows should be executed in parallel by multiple threads.

```
    #pragma omp parallel
    {
      structured block
    }
```

In this structured block, code that branches into or out of it is prohibited. The group of threads that execute the parallel block is called a *team*, the original thread is called the *master* and the additional threads are called *slaves* or *workers*. The number of threads started can be defined either by the user or by the system, and then the number of threads is typically equal to the number of available cores.

OpenMP follows the Fork/Join programming model (Figure 3.2). More analytically, program begins as a single process – master thread. The master thread executes sequentially until the first parallel region construct is encountered. When threads start their execution, they are forked by a process and when it is completed they join initial process.

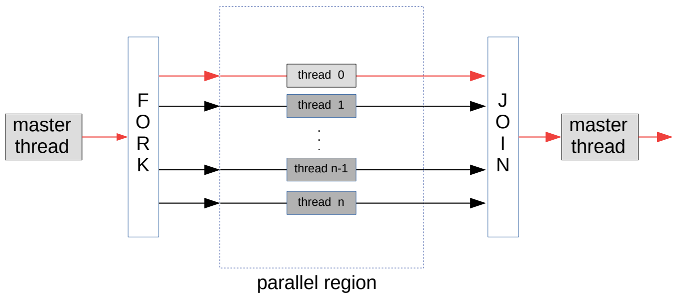

Figure 3.2: Illustration of Fork/Join model. Master–thread is represented by a red colored arrow and worker–threads by black colored arrows.

In the case where a block of code (that exists in a parallel region) must be executed only once by a single thread of the team, then a single directive is provided (`#pragma omp single`). If this single thread has to be also the master thread, OpenMP provides the master directive `#pragma omp master`. Note that there is no implied barrier, either on entry to or exit from the master section. So, in order to synchronise threads that belong in the same team, an explicit barrier is needed. OpenMP provides `#pragma omp barrier` and, when a thread encounters a barrier, it blocks until all threads that belong to the same team reach that barrier.

Since threads that belong to the same team share their resources, variables that are declared outside the parallel region are by default shared (shared scope). On the other hand, variables declared inside the parallel region are private to each thread (private scope). This can be changed using clauses `shared()` and `private()`.

When multiple threads try to access and simultaneously change a shared variable, a race condition occurs; for example, reduction

```
result = local_result + result;
```

inside a parallel block. This problem can be solved using clause `reduction(<operator>: <variable list>)`. Adding this clause in a parallel region, one can define the desired reduction operator $(+\,,*\,,-\,,/\,,\&\,,|\,,\hat{}\,,\&\&\,,||)$ and one or more reduction variables. OpenMP creates a private variable for each thread and, at run–time, system stores each thread's result in this variable. Also, a critical section is created and when the threads are done, all values stored in private variables are combined into a single shared reduction variable.

Except of `parallel` directive, there is also `parallel for` directive, which forks a team of threads to execute a block of code, that begins with a for loop. Iterations are divided among the additional threads. Usually, block partitioning is selected, which means that, for a total of n iterations and t number of threads, each thread executes $\frac{n}{t}$ iterations. Note that the loop variable is by default private (each thread has its own copy) and its value is updated in the same way as reduction variables.

Assignment to threads can have a very significant effect on performance. This can be achieved through scheduling. Schedule clause modifies the default scheduling option and has the form `schedule(<type>, <chunksize>)`. "Chunk "refers to a block of iterations that would be executed consecutively in the serial loop and chunksize is the number of iterations in the block, defined by a positive integer. Type can be {static, dynamic, guided, auto, runtime} and chunksize is optional. In static scheduling, iterations can be assigned to threads, before the loop is executed, following a round–robin scheduler. In dynamic and in guided scheduling, the loop iterations are assigned to each thread during execution. In both dynamic and guided scheduling, each thread executes a chunk and when it finishes, it requests another one from the run–time system. However, dynamic uses fixed size chunks and guided uses chunks that have a decreasing chunksize as the number of iteration increases. Runtime scheduling can be defined at runtime using specific environment variables and is useful when the cost of each iteration cannot be determined.

OpenMP on NUMA systems

In NUMA systems and, more specifically, in cache coherent NUMA systems (cc–NUMA), performance can also be affected by thread and data placement [11]. Selecting the optimal thread placement depends not only on the topology, but also on the characteristics of desired application. Threads can be placed either close (same socket) or far (different sockets). Putting threads far apart may improve memory bandwidth, combined cache size available for the application but can decrease performance on synchronization. In contrast, placing threads close may improve performance of synchronization constructs, but may decrease the available memory bandwidth and cache size. In order to understand how caches and threads are correlated, we present a short background.

Caches are faster than main memory and their design takes into consideration the principles of temporal and spatial locality. When a processor needs to access a location in main memory, rather than transferring data only from this location, a block of memory containing data from nearby locations is transferred to or from the processor. This block of memory is called a cache line (or cache block). Now cache coherent systems ensure that caches on different sockets are coherent. So, when multiple threads try to update different data that belong to the same cache line, the cache coherent control acts as if the threads were accessing the same memory location. This means that when one thread updates its data and others try to read them, they will have to retrieve them from main memory. This is called false sharing and can add extra latency, degrading the performance of a shared–memory program.

Thread placement can be achieved with OpenMP clauses and routines, but data placement on the other hand cannot. For that reason, one can use an other alternative, library libnuma. This library is developed for NUMA control and can be combined with OpenMP. This library is included in header file `<numa.h>`.

3.3 Parallel Implementation of CPD–ALS and RBS–CPD using ALS

In this section, we will describe a parallel implementation of ALS and RBS. We used OpenMP in order to execute both algorithms in parallel. Matrix operations are implemented using routines of the C++ library Eigen. Some Eigen's algorithms can exploit multi threading using OpenMP. We preferred to enable this feature only in sections that are outside of the parallel blocks, in order to have more control on resources.

3.3.1 CPD–ALS

In Chapter 2, we presented the ALS algorithm (Algorithm 1) without mentioning how it can be parallelized. In distributed systems, a parallel implementation of ALS can be similar to parallel NTF (Nonnegative Tensor Factorization) implementation, proposed in [5]. More specifically, a third–order tensor $\mathcal{X}$ can be partitioned into p sub–tensors, where p is the number of processing elements and can be factorized as $p = p_A \times p_B \times p_C$, forming a three–dimensional Cartesian grid. Each factor is also partitioned into blocks of rows, p_A, p_B, p_C respectively. Certain communication groups between processors are created and used for the efficient collaborative implementation of specific computational tasks. However, in a shared–memory system this type of communication is not needed, since all threads can access the same main memory. Therefore we followed a different approach to solve ALS in parallel. We tried to develop a parallel version of this algorithm, capable to run with equally high performance across most shared–memory systems.

In Chapter 2, we showed that there are four steps that include the main bottlenecks of ALS. Those are:

- update factor **A**,
- update factor **B**,
- update factor **C**,
- acceleration step.

At the first three steps, both Hadamard and Khatri–Rao products are computed very fast ($\approx 1\%$ the total time spent for each step). Most of the time is spent to compute each MTTKRP matrix **W** and multiplication with matrix **Z**, where:

$$\begin{aligned}
\mathbf{W_A} &= \mathbf{X}_{(1)}(\mathbf{B}\odot\mathbf{C}) \in \mathbb{R}^{I\times R}, \quad \mathbf{Z_A} = (\mathbf{B}^T\mathbf{B}\circledast\mathbf{C}^T\mathbf{C})^{-1} \in \mathbb{R}^{R\times R},\\
\mathbf{W_B} &= \mathbf{X}_{(2)}(\mathbf{C}\odot\mathbf{A}) \in \mathbb{R}^{J\times R}, \quad \mathbf{Z_B} = (\mathbf{C}^T\mathbf{C}\circledast\mathbf{A}^T\mathbf{A})^{-1} \in \mathbb{R}^{R\times R},\\
\mathbf{W_C} &= \mathbf{X}_{(3)}(\mathbf{B}\odot\mathbf{A}) \in \mathbb{R}^{K\times R}, \quad \mathbf{Z_C} = (\mathbf{B}^T\mathbf{B}\circledast\mathbf{A}^T\mathbf{A})^{-1} \in \mathbb{R}^{R\times R}.
\end{aligned} \tag{3.1}$$

Each matricized Tensor can be expressed as follows:

$$\begin{aligned}
\mathbf{X}_{(1)} &= \begin{bmatrix}\mathcal{X}_{(:,1,:)} & \dots & \mathcal{X}_{(:,J,:)}\end{bmatrix} = \begin{bmatrix}\mathbf{X}^1_{(1)}\dots\mathbf{X}^J_{(1)}\end{bmatrix}, \quad \mathbf{X}^j_{(1)} \in \mathbb{R}^{I\times K}\\
\mathbf{X}_{(2)} &= \begin{bmatrix}[\mathcal{X}_{(:,:,1)}]^T\dots[\mathcal{X}_{(:,:,K)}]^T\end{bmatrix} = \begin{bmatrix}\mathbf{X}^1_{(2)}\dots\mathbf{X}^K_{(2)}\end{bmatrix}, \quad \mathbf{X}^k_{(2)} \in \mathbb{R}^{J\times I}\\
\mathbf{X}_{(3)} &= \begin{bmatrix}[\mathcal{X}_{(:,1,:)}]^T\dots[\mathcal{X}_{(:,J,:)}]^T\end{bmatrix} = \begin{bmatrix}\mathbf{X}^1_{(3)}\dots\mathbf{X}^J_{(3)}\end{bmatrix}, \quad \mathbf{X}^j_{(3)} \in \mathbb{R}^{K\times I}.
\end{aligned} \tag{3.2}$$

The Khatri–Rao product can be written as:

$$(\mathbf{B}\odot\mathbf{C}) = \begin{bmatrix}\mathbf{B}_{(1,:)}\odot\mathbf{C}\\ \vdots\\ \mathbf{B}_{(J,:)}\odot\mathbf{C}\end{bmatrix}, \tag{3.3}$$

$$(\mathbf{C}\odot\mathbf{A}) = \begin{bmatrix}\mathbf{C}_{(1,:)}\odot\mathbf{A}\\ \vdots\\ \mathbf{C}_{(K,:)}\odot\mathbf{A}\end{bmatrix}, \tag{3.4}$$

$$(\mathbf{B}\odot\mathbf{A}) = \begin{bmatrix}\mathbf{B}_{(1,:)}\odot\mathbf{A}\\ \vdots\\ \mathbf{B}_{(J,:)}\odot\mathbf{A}\end{bmatrix}. \tag{3.5}$$

Using eq. (3.2 - 3.5), each matrix **W** can be computed as :

$$\begin{aligned}
\mathbf{W}_A &= \sum_{j=1}^{J}\mathbf{X}^j_{(1)}(\mathbf{B}_{(j,:)}\odot\mathbf{C})\\
\mathbf{W}_B &= \sum_{k=1}^{K}\mathbf{X}^k_{(2)}(\mathbf{C}_{(k,:)}\odot\mathbf{A})\\
\mathbf{W}_C &= \sum_{j=1}^{J}\mathbf{X}^j_{(3)}(\mathbf{B}_{(j,:)}\odot\mathbf{A}).
\end{aligned} \tag{3.6}$$

At the acceleration step, in order to compute function $f_{\mathcal{X}}$ for the new factors, we need to compute the corresponding MTTKRP:

$$\mathbf{W}_C^{accel} = \sum_{j=1}^{J} \mathbf{X}_{(3)}^{j} (\mathbf{B}_{(j,:)}^{accel} \odot \mathbf{A}^{accel}) \tag{3.7}$$

Each term can be computed in a parallel for loop, since there are no dependencies. More specifically, in a parallel for loop, each available thread can compute its own term and when it completes its task, the result is reduced using reduction clause, as described before. In case of a NUMA system, each N–mode matricized tensor and the respective Khatri–Rao products are partitioned in s blocks, where s is the number of available sockets. The main goal here is to exploit locality and increase cache usage as much as possible. In few words, threads in each socket compute their partial sum and when it is completed, the result is reduced. Then, also in parallel, only threads that are in the same socket with master thread, compute the product $\mathbf{WZ}$ and update each factor respectively. In both parallel for loops, the static scheduling clause was selected, since all iterations have equal cost. The parallel section of ALS that is related to factor update is given in Algorithm 3.

Algorithm 3 Parallel CPD-ALS algorithm

```
1: procedure PARALLEL CPD-ALS($\mathcal{X}$,$R$)
2:     initialize $k$, $f_{\mathcal{X}}$ and matrix factors $\mathbf{A}_0, \mathbf{B}_0, \mathbf{C}_0$
3:     while terminating condition is FALSE do
4:         $\mathbf{Z_A} = (\mathbf{B}^T\mathbf{B} \circledast \mathbf{C}^T\mathbf{C})^\dagger$
5:         In parallel, for $b = 1, ..., s$, do
6:             $\mathbf{W}_A^b = \sum_{j=(b-1)\frac{J}{s}+1}^{b\frac{J}{s}} \mathbf{X}_{(1)}^j(\mathbf{B}_{(j,:)} \odot \mathbf{C})$
7:         end parallel for
8:         $\mathbf{W}_A = \sum_{b=1}^{s} \mathbf{W}_A^b$
9:         $\mathbf{A}_{k+1} = \mathbf{W}_A\mathbf{Z}_A$
10:        $\mathbf{Z_B} = (\mathbf{C}^T\mathbf{C} \circledast \mathbf{A}^T\mathbf{A})^\dagger$
11:        In parallel, for $b = 1, ..., s$, do
12:            $\mathbf{W}_B^b = \sum_{j=(b-1)\frac{K}{s}+1}^{b\frac{K}{s}} \mathbf{X}_{(2)}^k(\mathbf{C}_{(k,:)} \odot \mathbf{A})$
13:        end parallel for
14:        $\mathbf{W}_B = \sum_{b=1}^{s} \mathbf{W}_B^b$
15:        $\mathbf{B}_{k+1} = \mathbf{W}_B\mathbf{Z}_B$
16:        $\mathbf{Z_C} = (\mathbf{B}^T\mathbf{B} \circledast \mathbf{A}^T\mathbf{B})^\dagger$
17:        In parallel, for $b = 1, ..., s$, do
18:            $\mathbf{W}_C^b = \sum_{j=(b-1)\frac{J}{s}+1}^{b\frac{J}{s}} \mathbf{X}_{(3)}^j(\mathbf{B}_{(j,:)} \odot \mathbf{A})$
19:        end parallel for
20:        $\mathbf{W}_C = \sum_{b=1}^{s} \mathbf{W}_C^b$
21:        $\mathbf{C}_{k+1} = \mathbf{W}_A\mathbf{Z}_A$
22:        $k \leftarrow k + 1$
23:    end while
24:    return matrix factors $(\mathbf{A}_k, \mathbf{B}_k, \mathbf{C}_k)$
25: end procedure
```

3.3.2 RBS–CPD using ALS

As explained previously, the number of blocks per dimension in RBS is defined as Q_n. In each outer iteration, we sample Q_n samples and, therefore, we solve Q_n problems. Each problem can be solved independently, since blocks are not overlapping. This means that we can solve Q_n problems in parallel, assigning each one to a corresponding thread. If $Q_n > t$ (number of available threads), then Q_n problems are divided in task-groups, and each thread executes $\frac{Q_n}{t}$ tasks sequentially.

For large–scale tensors, we expect that, using small Q_n (large block sizes), computational tasks can be performed significantly well in parallel. However, the number of data accesses is very high, which causes overhead during sampling. In contrast, for large Q_n, the computational cost decreases dramatically and even though that the number of data accesses is lower, sampling remains a bottleneck since it is very difficult to achieve locality.

The parallelized version of RBS using ALS is shown in Algorithm 4.

Algorithm 4 Parallel RBS–CPD algorithm

```
1: procedure RBS–CPD($\mathcal{X}$)
2:     initialize $k$ and matrix factors $\mathbf{A}, \mathbf{B}, \mathbf{C}$
3:     while no convergence do
4:         $l = 1$
5:         if $k > 1$ then
6:             Shuffle indices $I_n$
7:         end if
8:         In parallel, for $thread = 1...t$, do
9:             for $l = 1...\frac{Q_n}{t}$, do
10:                Generate $\mathcal{B}_n^l$ for $n = 1, 2, 3$
11:                Sample $\mathbf{A}_{sub}^{(l)}, \mathbf{B}_{sub}^{(l)}, \mathbf{C}_{sub}^{(iter)}, \mathcal{X}_{sub}^{(l)}$
12:                new $\mathbf{A}_{sub}^{(l)} \leftarrow$ update$\left(\mathcal{X}_{sub(1)}^{(l)}, \mathbf{B}_{sub}^{(l)}, \mathbf{C}_{sub}^{(l)}\right)$
13:                new $\mathbf{B}_{sub}^{(l)} \leftarrow$ update$\left(\mathcal{X}_{sub(2)}^{(l)}, \mathbf{C}_{sub}^{(l)}, \text{new } \mathbf{A}_{sub}^{(l)}\right)$
14:                new $\mathbf{C}_{sub}^{(l)} \leftarrow$ update$\left(\mathcal{X}_{sub(3)}^{(l)}, \text{new } \mathbf{B}_{sub}^{(l)}, \text{new } \mathbf{A}_{sub}^{(l)}\right)$
15:                $\mathbf{A}_{k+1}(\mathcal{B}_n^l, :) = \text{new } \mathbf{A}_{sub}^{(l)}$
16:                $\mathbf{B}_{k+1}(\mathcal{B}_n^l, :) = \text{new } \mathbf{B}_{sub}^{(l)}$
17:                $\mathbf{C}_{k+1}(\mathcal{B}_n^l, :) = \text{new } \mathbf{C}_{sub}^{(l)}$
18:            end for
19:        end parallel for
20:        $k \leftarrow k + 1$
21:    end while
22:    return matrix factors $\mathbf{A}, \mathbf{B}, \mathbf{C}$
23: end procedure
```

3.4 Numerical Experiments

In this section, we present results obtained from the parallel OpenMP implementation of ALS and RBS respectively.

3.4.1 Setup

Both programs are executed on ARIS (Advanced Research Information System), a Greek supercomputer, deployed and operated by Greek Research and Technology Network (GRNET) [1]. ARIS consists of 532 computational nodes separated in four "islands". The one that we used is called "fat node"and consists of a DELL PowerEdge R820 system with processor type Sandy Bridge - Intel(R) Xeon(R) CPU E5-4650v2 (44 nodes in total - 4 sockets per node - 10 cores per socket) and 512 GB RAM per node.

3.4.2 Numerical Experiments

The following results are obtained using synthetic data. More precisely, each rank R tensor is generated from random factor matrices, each with uniform distribution $\mathcal{U}(-1, 1)$. Data presented in the following figures are obtained by repeating each experiment 5 times, using the same input data. Note that we focus on performance, rather than convergence. Hence we restrict the number of iterations to 10.

CPD–ALS

In Figure 3.3, we plot the execution time of CPD–ALS, terminated at 10^{th} iteration, for a $1200 \times 1200 \times 1200$ tensor using $n = 1, 4, 8, 20, 40$ threads. In Figure 3.4 we plot the corresponding speedup. Speedup measures the relative performance between two systems processing the same problem, and is defined as

$$S = \frac{T_1}{T_2}, \tag{3.8}$$

where T_1 and T_2 denote the execution time of each system. More precisely, each S_t is calculated as $S_t = T_1/T_t$, where T_t denotes the execution time of a parallel implementation using t number of threads and T_1 denotes the execution time of a serial implementation $(t = 1)$.

In Figure 3.5, we plot the execution time of CPD–ALS, implemented in two versions, one NUMA aware and one naive. For a small number of threads, less than 8, both versions performed well, but as the number of threads increases, the NUMA aware version performs bettter. The corresponding speedup is illustrated in Figure 3.6.

[1]Greek Research and Technology Network - High Performance Computing Services `https://hpc.grnet.gr`

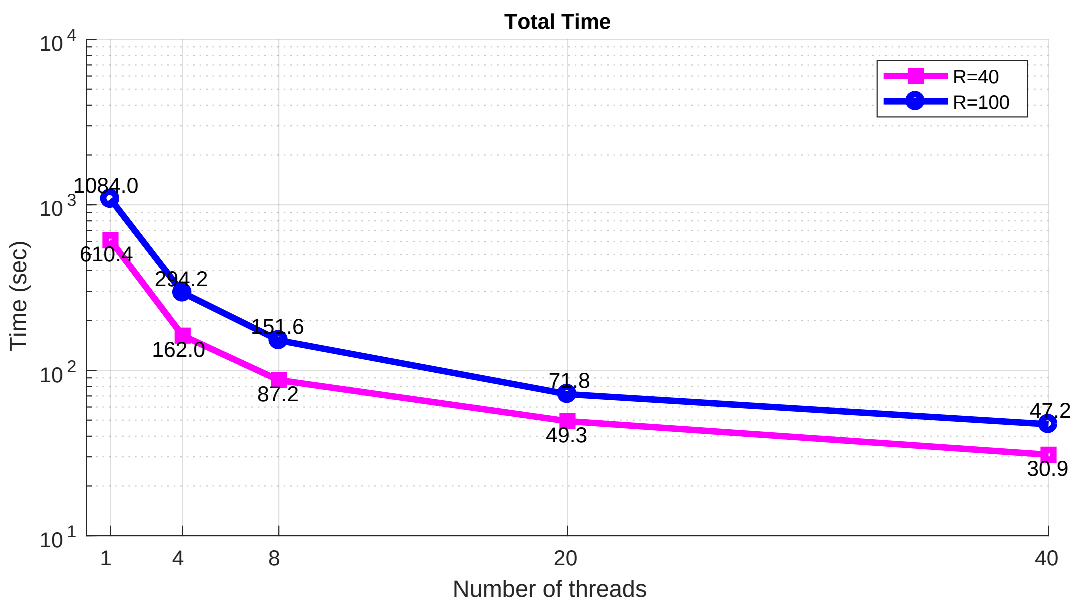

Figure 3.3: Execution time of CPD–ALS, terminated at 10^{th} iteration, for a $1200 \times 1200 \times 1200$ tensor using n threads, where $n = 1, 4, 8, 20, 40$.

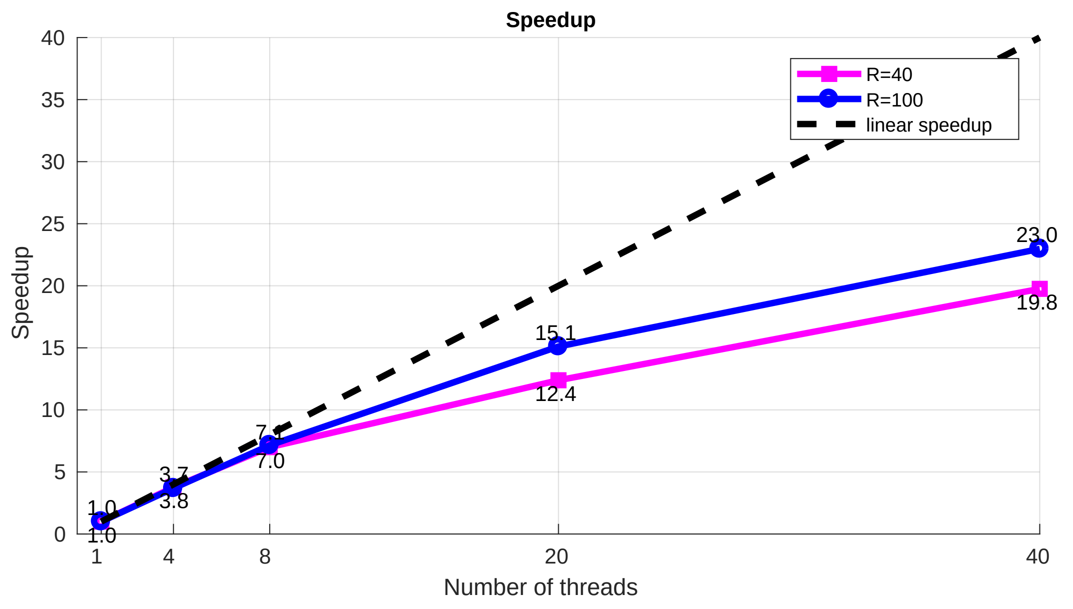

Figure 3.4: Speedup of CPD–ALS, terminated at 10^{th} iteration, achieved for a $1200 \times 1200 \times 1200$ tensor using n threads, where $n = 1, 4, 8, 20, 40$.

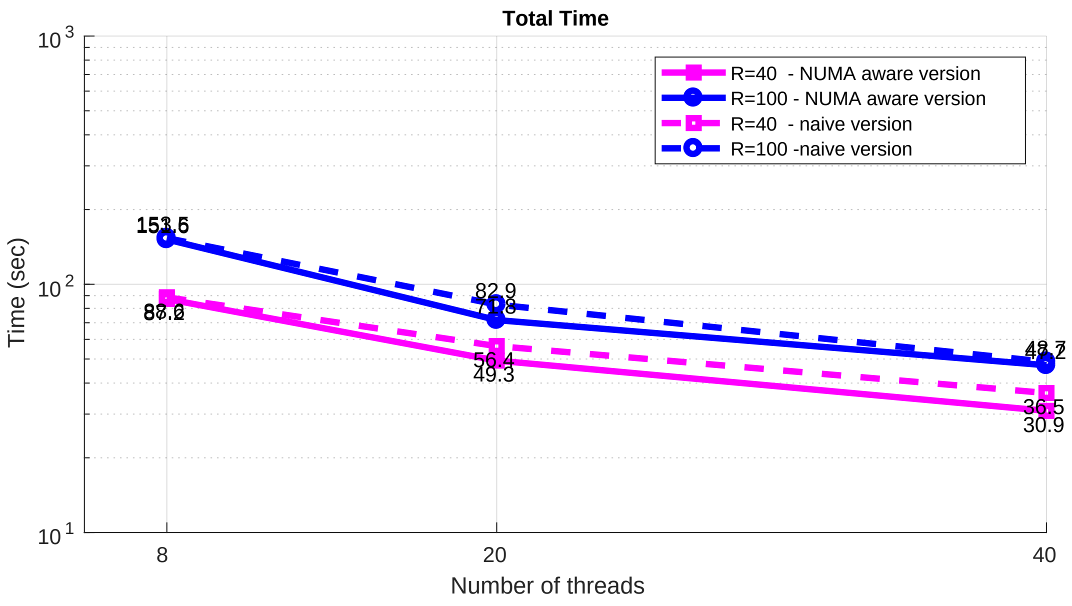

Figure 3.5: Execution time of CPD–ALS using a parallel NUMA aware version vs. a parallel naive version, using $n = 8, 20, 40$ threads.

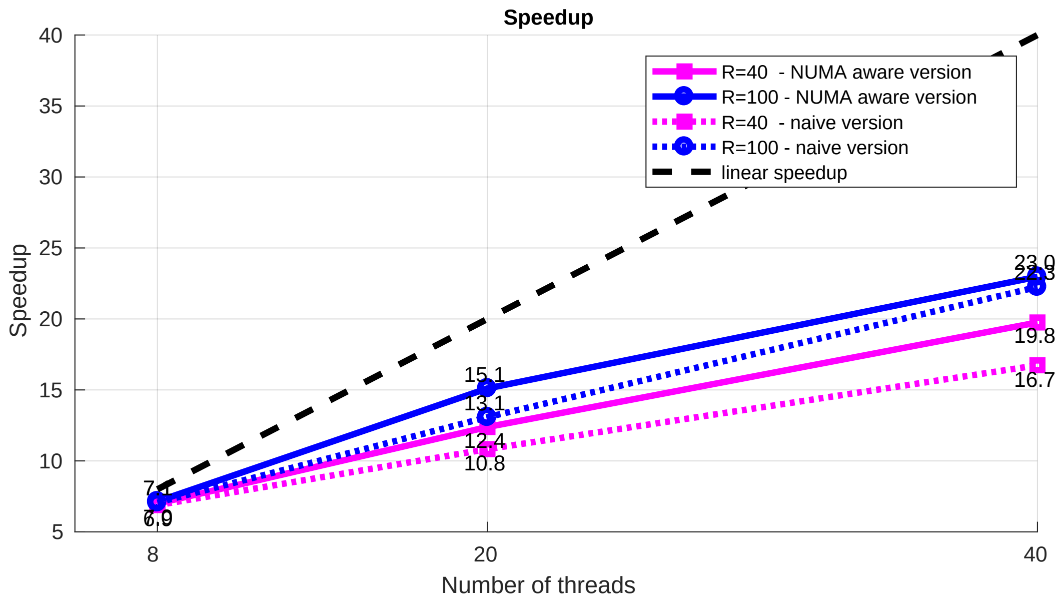

Figure 3.6: Speedup of CPD–ALS using a parallel NUMA aware version vs. a parallel naive version, using $n = 8, 20, 40$ threads.

RBS–CPD using ALS

The following results are obtained using a $2000 \times 2000 \times 2000$ tensor with rank $R = 20$. In Figure 3.7 we plot the execution time of RBS–CPD for several values of Q_n, using 1 thread. In Figure 3.8 we plot the relative factor change during the first 10 iterations, for the aforementioned values of Q_n. We observe that for larger block sizes ($Q_n = 1, 2, 4, 8$), RBS–CPD achieves lower relative factor change.

In Figures 3.9, 3.10 we plot the execution time of block sizes (Q_n = 4, 8, block size B = 500, 250) which attain the best speedup, while in Figures 3.11, 3.12 we plot the execution time of block sizes (Q_n = 40, 80, block size B = 50, 25) which attain the worst speedup.

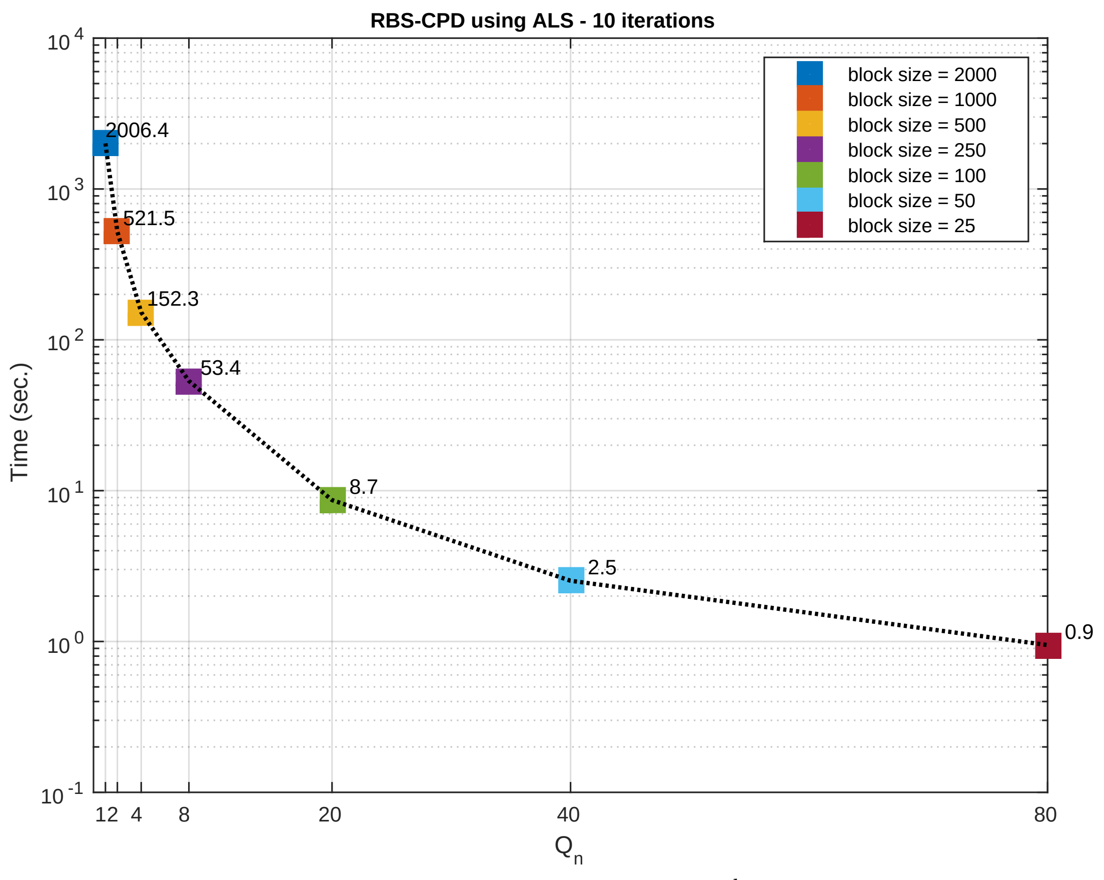

Figure 3.7: Execution time of RBS–CPD, terminated at 10^{th} iteration, for a $2000 \times 2000 \times 2000$ tensor and rank $R = 20$. We solve $Q_n = 1, 2, 4, 8, 20, 40$ inner problems, using 1 thread.

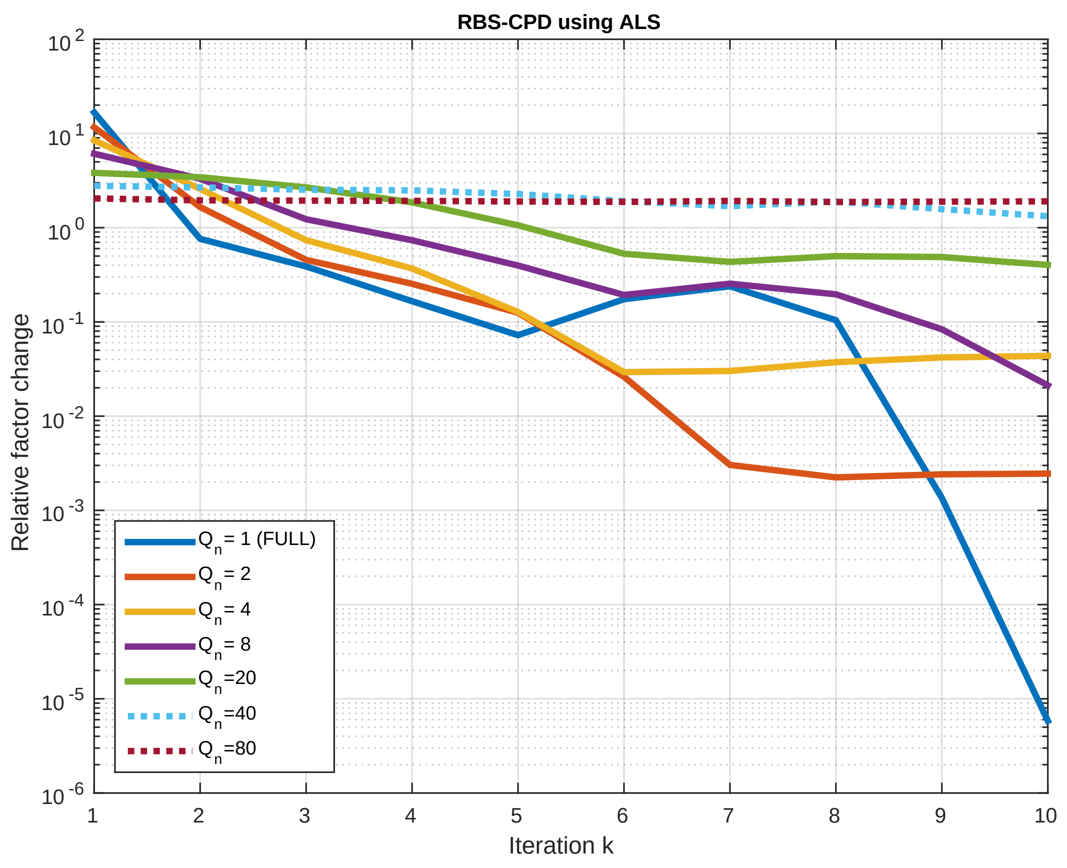

Figure 3.8: Plot of relative factor change during the first 10 iterations of RBS-CPD.

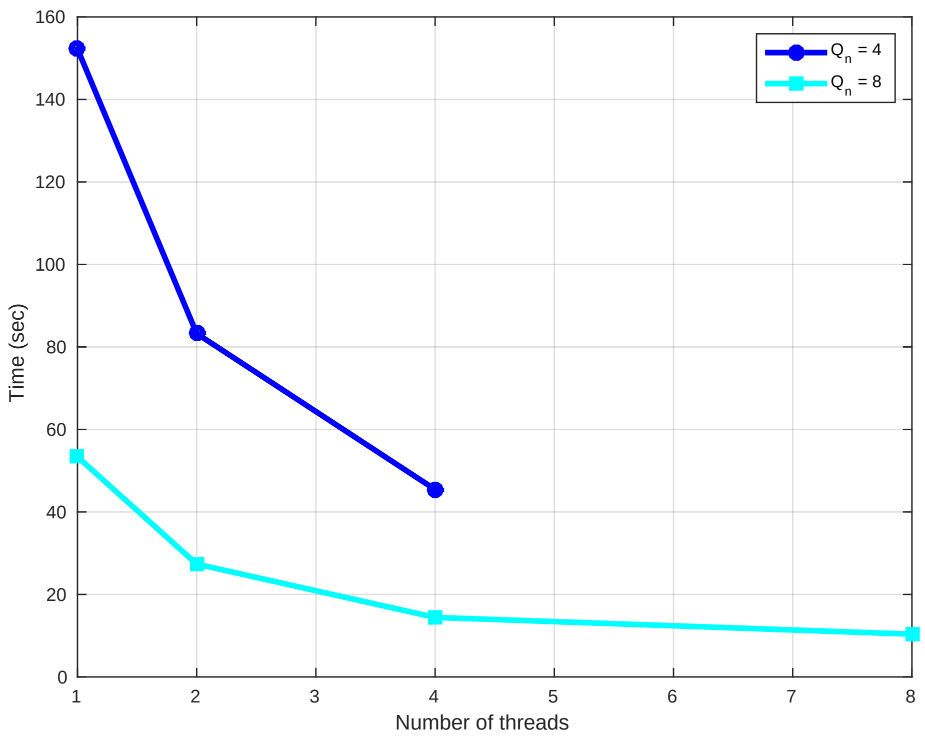

Figure 3.9: Execution time of RBS–CPD, terminated at 10^{th} iteration, for a $2000 \times 2000 \times 2000$ tensor, solving $Q_n = 4, 8$ inner problems.

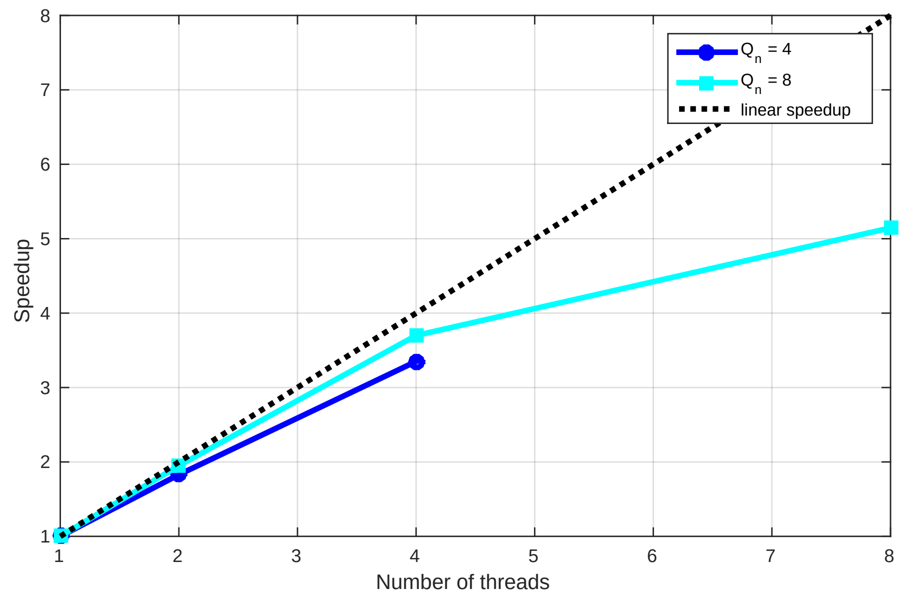

Figure 3.10: Speedup of RBS–CPD, terminated at 10^{th} iteration, for a $2000 \times 2000 \times 2000$ tensor, solving $Q_n = 4, 8$ inner problems.

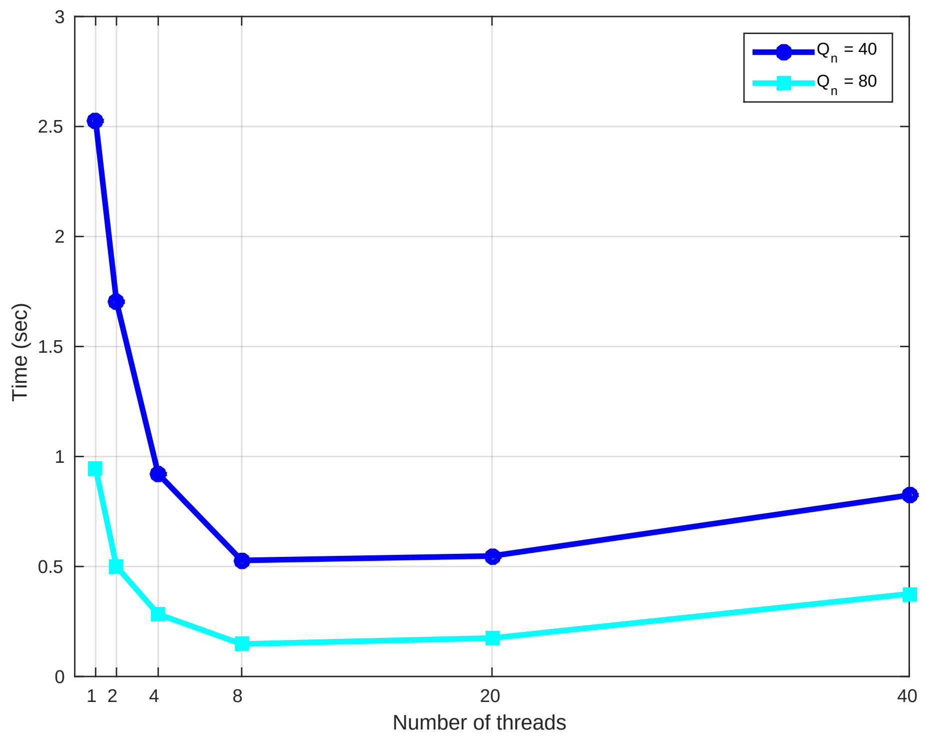

Figure 3.11: Execution time of RBS–CPD, terminated at 10^{th} iteration, for a $2000 \times 2000 \times 2000$ tensor, solving $Q_n = 40, 80$ inner problems

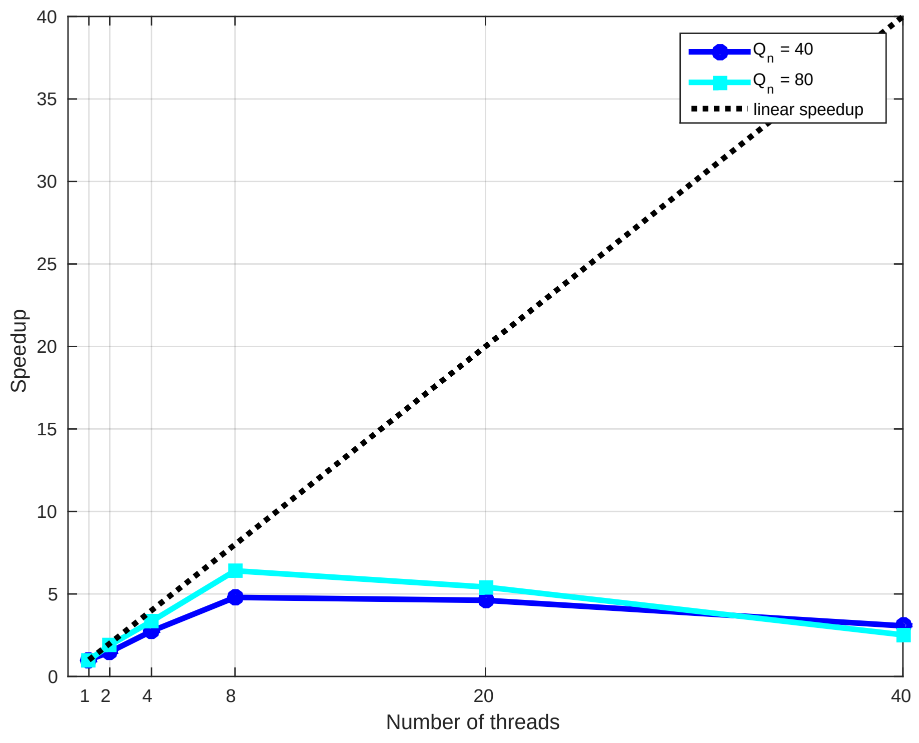

Figure 3.12: Speedup of RBS–CPD, terminated at 10^{th} iteration, for a $2000 \times 2000 \times 2000$ tensor, solving $Q_n = 40, 80$ inner problems.

Chapter 4

Parallel Implementation using CUDA

For many years, computers contained only general–purpose CPUs designed to run general programming tasks. In the late 80's, new graphic interfaces and 2D applications led in development of special display accelerators that replaced simple video display cards. Almost one decade later, the first generation of graphics processing units came out and few years later General Purpose Computing GPUs (GPGPUs) were introduced. Since their first appearance, GPGPUs have become more powerful and have begun making computational inroads against the CPUs in many fields, such as 3D applications, linear algebra and, more recently, in machine learning.

4.1 Introduction to Heterogeneous Computing

A modern heterogeneous computational node consists of multicore CPU sockets and hardware accelerators such as GPUs or even FPGAs. The most common hardware accelerators are GPUs, used to accelerate the execution of a program section. A GPU is currently not a stand-alone platform but works as a co–processor to a CPU. Therefore, GPUs must operate in conjunction with a CPU-based host through a PCI–Express bus. For that reason, the CPU is called the *host* and the GPU is called the *device* (Figure 4.1).

The code of a heterogeneous application consists of two parts, the *host code* and *device code* or *kernel.* Host code runs on CPUs and device code runs on GPUs. In a program implemented on Heterogeneous Platforms, the host is responsible for data initialization, synchronization across all devices, managing the environment, code and data for the device before loading tasks on the device.

It is important to mention that GPU computing is not meant to replace CPU computing, since each approach has advantages for certain kinds of programs. CPU computing is good for control–intensive tasks and, on the other hand, GPU computing is good for data-parallel computation-intensive tasks. Furthermore, if a problem has a small data size, complicated control logic and low level parallelism, then CPU is a good choice due to its ability to handle heavy weighted tasks. On the other hand, if the problem has a large data size and exhibits massive data parallelism, then the GPU is more suitable, since it can support massive multi–threading and has larger bandwidth than a CPU. More details about their differences will be presented in the following section and also a background on the GPU architecture is given. Figure 4.2 illustrates the hierarchy and relation between components.

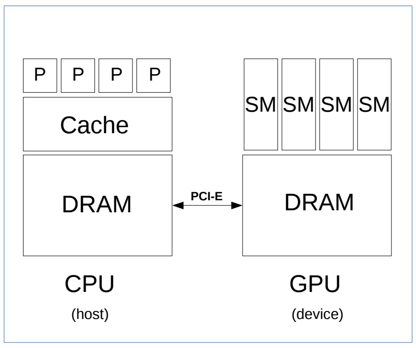

Figure 4.1: Illustration of Heterogeneous architecture.

4.2 GPU Architecture

The architecture of general–purpose computing GPUs is classified as *Single Instruction – Multiple Threads*(SIMT). This means that multiple threads are organized in groups and execute the same instruction. However, each thread has its own instruction address counter, register state and carries out the current instruction on its own data. Unlike CPUs, which have multiple large cores and where each core is designed with a complicated control, optimized to execute sequential tasks, GPUs have many cores (often hundreds or even thousands) smaller in size, with a simpler control and are ideal for data–parallel tasks. Also, GPU cores can handle threads more easily than the CPU does, due to the fact that they are extremely light–weight and are organized in groups, making them more easy to schedule.

The key component of a GPU is the *Streaming Multiprocessor* (SM). Using multiple copies of this building block, GPU hardware parallelism is achieved. Each SM consists of many cores, where each core executes one corresponding thread. SM is designed to support concurrent execution of hundreds of threads, and since there are multiple SMs on a single GPU, it is possible to execute thousands of threads concurrently. All threads that belong to the same SM are called a *thread block* and if the number of threads in one block is equal to 32, which is the optimal in terms of performance, then this block is also called as a *warp*. All threads that belong to the same device are called *grid*.

The SIMT architecture is similar to the *Single Instruction - Multiple Data* (SIMD) architecture, was the basis for vector supercomputers. Both architectures implement parallelism by broadcasting the same instruction to multiple execution units. However, one main difference is that SIMD requires that all elements in a vector execute together in a unified synchronous group, whereas SIMT allows multiple threads in the same warp to execute independently. In other words, even though all threads in a warp start simultaneously at the same program address, it is possible for individual threads to have different behaviour (e.g. branches).

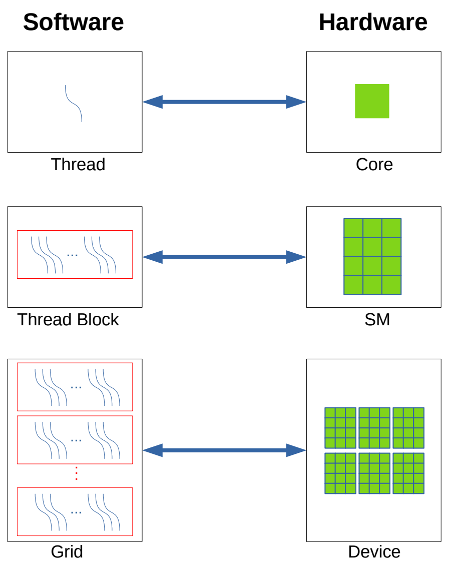

Figure 4.2: Correspondence between Software & Hardware components.

4.3 CUDA C/C++

CUDA is a parallel computing platform and API, with a small set of extensions to the C/C++ language and is developed by NVIDIA [12]. Using CUDA as a programming model, one can execute applications on heterogeneous systems that consists of CPUs and NVIDIA CUDA-enabled GPUs only. This means that CUDA is not compatible with different setups, unlike other frameworks, like OpenCL, which is an open standard without hardware limitations. Despite those limitations, we prefered to use CUDA rather than other frameworks due to reasons that will be explained below.

CUDA provides both a low level API (CUDA Driver API) and a higher level API (CUDA Runtime API). Due to the higher complexity of the first one, we prefered to use the second one. It comes also with a wide variety of math libraries (cuBLAS for basic Linear Algebra applications, cuFFT for Fast Fourier Transform computations, cuSOLVER for dense and sparse operations, etc.) and other software components for compilation, debugging and performance improvement. Unlike OpenCL, where device code is compiled during run-time, CUDA toolkit offers a dedicated compiler, called `nvcc`. This compiler separates each code, sends host code to a C/C++ compiler and device code to the GPU. The device code (kernel) is further compiled by nvcc, is optimized according to the target GPU and then is linked to the main program.

4.3.1 CUDA programming model

As mentioned before, a C/C++ application developed in heterogeneous platform using CUDA consists of two parts, the host code and the kernel. Host code is written in ANSI C/C++ and kernel is written using CUDA C/C++, all in one ore in separated source files. A typical processing flow of a CUDA program has the following pattern. At first, GPU memory needs to be allocated and then data are copied from CPU memory to GPU memory. Then, host invokes kernel execution on device and waits until execution is completed. When it finishes, data from GPU are copied back to CPU memory.

4.3.2 CUDA Memory Management

Memory management in CUDA programming is similar to C programming, with the added programmer responsibility of explicitly managing data movement between the host and device. CUDA runtime provides functions to allocate, transfer and deallocate device memory.

Memory allocation:

In order to allocate global memory on the host, one can use the following function:

```
cudaError_t cudaMalloc(void **devPtr, size_t count);
```

where `count` is the number of bytes to be allocated in global memory on the device and `devPtr` the pointer of the location in which data are allocated. In the case of failure, the cudaMalloc function returns `cudaErrorMemoryAllocation`.

Memory deallocation:

The same allocated memory can be freed when is not needed using the function:

```
cudaError_t cudaFree(void *devPtr);
```

This function can fail if `devPtr` is not valid or if memory is already freed. It is important to mention that both operations are expensive and for that reason, one should use them as less frequently as possible.

Memory transfer:

Once global memory is allocated, data can be transferred from host to the device through the following function:

```
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count,
                       enum cudaMemcpyKind kind);
```

where `src` is the source memory location, `dst` is the destination memory location and `count` is the number of bytes to be transferred. Enumerated variable `kind` specifies the direction of the copy and its value can be one of the following:

`cudaMemcpyHostToHost`	Host $\rightarrow$ Host
`cudaMemcpyHostToDevice`	Host $\rightarrow$ Device
`cudaMemcpyDeviceToHost`	Device $\rightarrow$ Host
`cudaMemcpyDeviceToDevice`	Device $\rightarrow$ Device

4.3.3 CUDA kernel functions

Kernel functions in CUDA use the prefix `__global__` which is placed before the type definition (e.g. `__global__ <type> <function_name>(<function_parameters)>`). When a kernel function is launched from the host side, execution is moved to a device where a large number of threads are generated and each thread executes the statements specified by the kernel function.

Kernel functions are called by the host, in a similar way to ANSI C/C++. The only difference is that host also defines the grid-size and the block-size using `<<<, >>>` symbols, according to the following syntax:

```
kernel_name <<<gridsize, blocksize>>>(parameters);
```

4.3.4 Thread and Block Hierarchy

All threads in a grid share the same global memory space, all threads in a block can cooperate with each other using block-local synchronization and shared memory, and each thread has its own private register. Threads rely on two unique coordinates to distinguish themselves from each other, variable `blockIdx` (block index within a grid) and variable `threadIdx` (thread index within a block). Both are of type uint3, a CUDA built-in vector type that contains fields `x`, `y`, `z` and are accessed as `threadIdx.x (threadIdx.y , threadIdx.z)` and `blockIdx.x (blockIdx.y, blockIdx.z)`. Also, block-size and grid-size are defined using two `uint3` type variables, `blockDim` and `gridDim` which are accessed similar to indices. Separate threads in different blocks can be distinguished using a unique global thread_id, which is calculated using coordinate variables

`thread_id_dim = blockIdx.dim*blockDim.dim + threadIdx.dim`, where `dim={x,y,z}`.

4.3.5 CUDA Streams

In most cases, more time is spent to execute the kernel than transferring the data. One can exploit this situation through CUDA streams and may be able to hide the CPU–GPU latency. A CUDA stream refers to a sequence of asynchronous CUDA operations that execute on a device in the order issued by the host code. A stream encloses these operations, maintains their ordering, allows operations to be queued in the stream to be executed after all preceding operations and also, allows for querying the status of queued operations. These operations can include host–device data transfer, kernel launches, and other commands that are issued by the host but handled by the device. The execution of an operation in a stream is always asynchronous with respect to the host. In other words, the functions in the CUDA runtime API, can be classified as either *synchronous* or *asynchronous*. Functions with *synchronous* behavior block the host thread until they complete. On the other hand, functions with *asynchronous* behavior return control to the host immediately after being called. Asynchronous functions may require extra synchronization with the host, which can be accomplished using `cudaDeviceSynchronize()`.

Also, since all CUDA stream operations are asynchronous, the CUDA API provides a blocking function

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
```

which forces the host to block until all operations in the provided stream have completed. When a stream is no longer in use, its resources can be released using function

```
cudaError_t cudaStreamDestroy(cudaStream_t stream); .
```

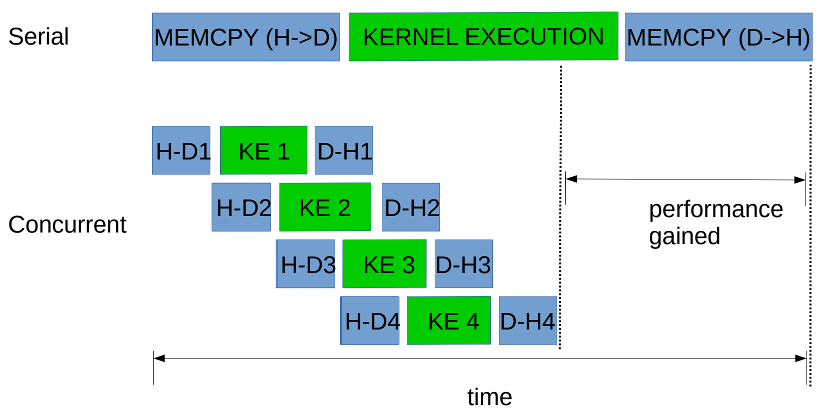

Figure 4.3: Timeline of a simple CUDA operation with 4 streams (concurrent) and without (serial).

4.3.6 Multi-GPU Parallelism

There are many reasons for adding multi-GPU support to an application. The most common reasons are:

1. increase available GPU memory (e.g. data sets are too large to fit into the memory of a single GPU),

2. increase throughput and efficiency (e.g. execute multiple tasks concurrently).

In order to use multi-GPU systems, it is important to understand the connection topologies. In a shared memory system, two or more GPUs can be connected either via PCI-E bus (through CPU), or via NVLink, a multi-lane near-range communication link (compatible only for NVIDIA GPUs), a communication protocol that offers higher bandwidth. Figure 4.4 illustrates the two types of a simplified GPU-CPU communication topology within a node.

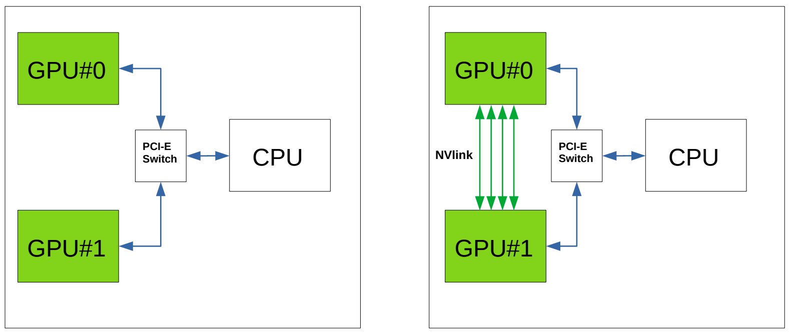

Figure 4.4: Illustration of a multi-GPU communication topology within a node. PCI-E based communication (left) vs NVLink based communication(right).

GPUs that belong to different nodes, can also communicate through computer-networking communication protocols (e.g. InfiniBand), but are beyond the scope of this thesis. Also, due to hardware limitations, we will only focus on PCI-E based communication.

Peer-to-Peer Communication

Kernels executed on modern GPUs can access directly the global memory of any GPU connected to the same PCIe root node. This can be achieved using CUDA *peer-to-peer* (P2P), which offers P2P Access and P2P Transfer directly between multiple GPUs. In case where this protocol is not supported, CUDA P2P API will perform peer-to-peer transfer between these devices, but the driver will transfer data through host memory for those transactions rather than directly across the PCIe bus. However this method adds extra latency in our application. In our implementation, we used the second approach due to hardware limitations.

Managing Multiple GPUs

Multiple GPUs in one node can be used at once from either one single thread (using non blocking methods) or multiple threads. Since most of the operations were executed synchronously, we used the second approach, where each host thread cooperates with each GPU using one stream. We used OpenMP in order to fork the necessary threads.

In general, in order to use multiple devices, the first step is to determine the number of CUDA-enabled devices that are available in a system using the following function:

```
cudaError_t cudaGetDeviceCount(int* count);.
```

If there are more than one CUDA compatible GPUs, one can select the current device with the following function:

```
cudaError_t cudaSetDevice(int id);
```

where id $\in [0, ..., count - 1]$. Once the current device is selected, all CUDA operations will be applied to that device as described before.

The next step is to allocate all resources needed in each device, in particular, create a stream and allocate memory on each device. Then, tasks are launched through the corresponding stream, one stream per device, and when the operations of each stream are completed, data are transferred back to the host. When the execution is completed, all allocated resources in each device must be freed. Algorithm 5 summarizes the above workflow.

Algorithm 5 multiGPU management

```
 1: Get number of GPUs ($count$)
 2: Fork $count$ threads
 3: **In parallel, for** $gpu\_id = 0, ..., count - 1$, **do**
 4:     Select current GPU with $id = gpu\_id$
 5:     Create stream for current GPU
 6:     Allocate device resources
 7:     Launch task(s) on current GPU through the corresponding stream
 8:     Use stream to synchronize devices
 9:     Destroy stream and free resources
10: **end parallel for**
```

4.4 Parallel Implementation of ALS

As we shown in Chapter 3, the main bottleneck of CPD is the computation of MTTKRP for each factor. In this section, we will present an alternative version of CPD, where we calculate MTTKRP using partial Khatri–Rao product, instead of the full product. The reason that we followed this approach is that the computation of full Khatri–Rao product does not perform better on a GPU than on a CPU. More specifically, the time spent on the computation can be compared to the time spent on copying the data to and from GPU. Also, the Hadamard product of small matrices (rank $R < 100$) is a low-performing operation on GPU since the memory bandwidth becomes the main bottleneck for such a small computational workload. That is why partial Khatri–Rao and Hadamard product are both computed on CPU. On the contrary, matrix multiplication performs much better on GPUs than on CPUs, especially after a certain threshold matrix size (it takes at least matrix sizes of around 100×100).

The main goal is to decompose each computational demanding operation into more, less computational expensive tasks. This leads to a more efficient GPU parallelization, since we can allocate significant less memory (e.g. $I \times K$ rather than $I \times J \times K$ in MTTKRP $\mathbf{W}_C$), exploit many optimization tricks such as CUDA Streams and also, achieve better multi-GPU parallelism. In this implementation of CPD-ALS, we tried to reuse part of the MTTKRP in order to update both factors $\mathbf{A}$ and $\mathbf{B}$. Later we will prove that both MTTKRPs $\mathbf{W}_A, \mathbf{W}_B$ can be computed using the same matrix $\mathbf{X}_{(1)}^{part}$ which is computed as follows.

$$\mathbf{X}_{(1)}^{part} = [\mathbf{X}_{(1)}^{1}C \quad ... \quad \mathbf{X}_{(1)}^{J}C]. \tag{4.1}$$

For the remainder of this thesis, we will refer to it as Partial MTTKRP.
Each MTTKRP is computed as shown below:

$$\begin{aligned}\mathbf{W}_A = \mathbf{X}_{(1)}(\mathbf{B} \odot \mathbf{C}) &= \sum_{j=1}^{J} \mathbf{X}_{(1)}^{j}[(\mathbf{B}_{(j,:)} \odot \mathbf{C})] \\ &= \sum_{j=1}^{J} [\mathbf{X}_{(1)}^{j} C] D_{\mathbf{B}_{(j,:)}} = \sum_{j=1}^{J} [\mathbf{X}_{(1)}^{part}]_{(:,(j-1)R:jR)} \mathbf{D}_{B_{(j,:)}},\end{aligned} \tag{4.2}$$

where $\mathbf{D}_{B_{(j,:)}} = diag\big(\mathbf{B}_{(j,:)}^{T}\big) \in \mathbb{R}^{R \times R}$. We remind to the reader that matrix $\mathbf{W}_B$ is defined as $\mathbf{W}_B = \mathbf{X}_{(2)}(C \odot A)$. Each row of $\mathbf{W}_B$ can be computed as

$$[\mathbf{W}_B]_{(j,:)} = \mathbf{1}^T([\mathbf{X}_{(1)}^{part}]_{(:,(j-1)R:jR)} \circledast \mathbf{A}), \tag{4.3}$$

where $\mathbf{1} = [1\,1\,...\,1\,1] \in \mathbb{R}^{I \times 1}$, and $\mathbf{W}_C$ is calculated as:

$$\begin{aligned}\mathbf{W}_C = \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A}) &= \sum_{j=1}^{J} \mathbf{X}_{(3)}^{j}(\mathbf{B}_{(j,:)} \odot \mathbf{A}) \\ &= \sum_{j=1}^{J} (\mathbf{X}_{(3)}^{j}\mathbf{A})\mathbf{D}_{B_{(j,:)}} = \sum_{j=1}^{J} \mathbf{X}_{(3)}^{j}(\mathbf{A}\mathbf{D}_{B_{(j,:)}}) \\ &= \sum_{j=1}^{J} \mathbf{B}_{(j,:)} \odot (\mathbf{X}_{(3)}^{j}\mathbf{A}).\end{aligned} \tag{4.4}$$

Proof of 4.3:

Since $\mathbf{W}_B = \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A}) = \sum_{k=1}^{K} \mathbf{X}_{(2)}^{k}(\mathbf{C}_{(k,:)} \odot \mathbf{A})$, each j^{th} row is equal to

$$\begin{aligned}[\mathbf{W}_B]_{(j,:)} = [\mathbf{X}_{(2)}]_{(j,:)}(\mathbf{C} \odot \mathbf{A}) &= \sum_{k=1}^{K} [\mathbf{X}_{(2)}^{k}]_{(j,:)}(\mathbf{C}_{(k,:)} \odot \mathbf{A}) \\ &\overset{(3.2)}{=} \sum_{k=1}^{K} [\mathcal{X}_{(:,j,k)}]^T(\mathbf{C}_{(k,:)} \odot \mathbf{A}) = \sum_{k=1}^{K} [\mathcal{X}_{(:,j,k)}]^T \mathbf{A}\mathbf{D}_{C_{(k,:)}} \\ &= \sum_{k=1}^{K} \mathbf{1}^T[\mathcal{X}_{(:,j,k)}\mathbf{C}_{(k,:)}] \circledast \mathbf{A} = \mathbf{1}^T\left(\sum_{k=1}^{K}[\mathcal{X}_{(:,j,k)}\mathbf{C}_{(k,:)}] \circledast \mathbf{A}\right) \\ &= \mathbf{1}^T\big([\mathcal{X}_{(:,j,:)}\mathbf{C}] \circledast \mathbf{A}\big) = \mathbf{1}^T\left([\mathcal{X}_{(1)}^{j}\mathbf{C}] \circledast \mathbf{A}\right) \\ &= \mathbf{1}^T\left([\mathcal{X}_{(1)}^{part}]_{(:,(j-1)R:jR)} \circledast \mathbf{A}\right) \qquad \blacksquare\end{aligned}$$

Depending on whether $K > I$ or not, we use the following formulas to compute matrix $\mathbf{W}_C$, if $K > I$:

$$\mathbf{W}_C = \sum_{j=1}^{J} \mathbf{X}_{(3)}^{j} (\mathbf{B}_{(j,:)} \odot \mathbf{A}), \tag{4.5}$$

otherwise,

$$\mathbf{W}_C = \sum_{j=1}^{J} \mathbf{B}_{(j,:)} \odot (\mathbf{X}_{(3)}^{j} \mathbf{A}). \tag{4.6}$$

Using this trick we can reduce Khatri–Rao product operations to $min(I \times R, K \times R)$ for each j-th term.

Based on the equations 4.1 - 4.6, operations that are performed on GPU are:

- full matrix $\mathbf{X}_{(1)}^{part}$,
- each term$[\mathcal{X}_{(1)}^{part}]_{(:,(j-1)R:jR)} D_{\mathbf{B}_{(j,:)}}$ of $\mathbf{W}_A$,
- matrix multiplications $\mathbf{B}^T\mathbf{B}$ and $\mathbf{C}^T\mathbf{C}$,
- factor update $\mathbf{A} = \mathbf{W}_A {\mathbf{Z}_A}^{-1}$,
- matrix multiplications $\mathbf{C}^T\mathbf{C}$ and $\mathbf{A}^T\mathbf{A}$,
- factor update $\mathbf{B} = \mathbf{W}_B {\mathbf{Z}_B}^{-1}$,
- matrix $\mathbf{W}_C$,
- matrix multiplications $\mathbf{B}^T\mathbf{B}$ and $\mathbf{A}^T\mathbf{A}$,
- factor update $\mathbf{C} = \mathbf{W}_C {\mathbf{Z}_C}^{-1}$,
- matrix $\mathbf{W}_C^{accel}$, used after acceleration step.

Matrix multiplications are implemented using two different methods, a low level implementation using *tiled matrix multiplication* and a high level using *cuda_Dgemm* from cuBLAS API. On the one hand, tiled matrix multiplication is a simple algorithm that exploits two very important factors compared to naive matrix multiplication. The first one is that more shared memory is used, which reduces global memory accesses and also, improves thread scheduling and execution, especially when tiles have the size of a warp (32×32).

Tiled Matrix Multiplication

More precisely, for matrices $\mathbf{A} \in \mathbb{R}^{M \times N}, \mathbf{B} \in \mathbb{R}^{N \times K}$ and $\mathbf{C} \in \mathbb{R}^{M \times K}$, where $\mathbf{C} = \mathbf{AB}$, each element $\mathbf{C}(i,j)$ is computed as:

$$\mathbf{C}(i,j) = \sum_{n=1}^{N} \mathbf{A}(i,n) \cdot \mathbf{B}(n,j) = \sum_{s=1}^{\frac{N}{|Tile|}} \sum_{t=1}^{|Tile|} \mathbf{A}(i,t) \cdot \mathbf{B}(t,j)\,. \tag{4.7}$$

Therefore, matrix $\mathbf{C}$ can be partitioned into square sub-matrix (tiles), all threads that belong to the same block compute one square sub-matrix and each thread an element of this sub-matrix. Threads access global memory using their global id, data from the global memory is then copied in the shared memory, and each thread inside that block uses its local id to access the corresponding cell. If the matrix dimensions are not divisible by the block dimensions, then extra operations are needed (such as zero-padding and extra control logic) in order to include the leftover elements, which causes performance degradation. Thus, this method is very efficient when the sizes of the matrix dimensions are multiples of the tile dimensions, and also, performs better than cuda_Dgemm for small matrices. In contrast, for large matrices, cuda_Dgemm achieves higher performance. As a result, we opt for the second method for our implementation.

Matrix Multiplication using function cuda_Dgemm()

Function cuda_Dgemm() (in general cuda_<type>gemm - D stands for double precision) implements Level-3 (matrix-matrix operations) Basic Linear Algebra Subprograms. This function performs the matrix–matrix multiplication as follows:

$$\mathbf{C} = \alpha \cdot op(\mathbf{A})op(\mathbf{B}) + \beta \cdot \mathbf{C}, \tag{4.8}$$

where α, β are scalars, $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are matrices in column-major format and operator $op(\mathbf{M}) = \{\mathbf{M} \text{ or } \mathbf{M}^T \text{ or } \mathbf{M}^H\}$. In order to use this function, an additional variable is needed (handle variable `cublasHandle_t`). The cublasHandle_t type is a pointer type to an opaque structure holding the cuBLAS library context. This variable must be initialized using `cublasCreate()` and should be destroyed at the end using `cublasDestroy()`. Creating a new handle can add extra overhead but, on the other hand, can be reused for multiple cuBLAS operations. Note that in a multi-GPU application multiple handles must be used, each one is created by a host thread, since each thread can only hold one GPU context.

Each sub-matrix of $\mathbf{X}_{(1)}^{part}$ can be computed separately using cuda_Dgemm(). The MTTKRPs $\mathbf{W}_C$ and $\mathbf{W}_C^{accel}$ are computed in a way similar to "tiled matrix multiplication", as presented below in Algorithm 6 and 7.

From all the above operations, three of them are the main bottleneck of the algorithm and therefore, due to their demanding complexity can be computed efficiently on multiple GPUs. Those are the computations of $\mathbf{X}_{(1)}^{part}$, $\mathbf{W}_C$ and $\mathbf{W}_C^{accel}$ which consist of 90% of the total execution time.

The Partial MTTKRP $\mathbf{X}_{(1)}^{part}$ as in (4.1) can be computed independently. Therefore, each part of $\mathbf{X}_{(1)}^{part}$, $\mathbf{X}_{(1)}^{j}C$ with $j \in \{1, ..., J\}$, can be assigned to the corresponding GPU as we show in Algorithm 8. MTTKRPs $\mathbf{W}_C$ and $\mathbf{W}_C^{accel}$ can be partitioned in as many blocks as the number of available GPUs. Each block is calculated independently and when the procedure is completed, all block results are gathered and summed, as presented in Algorithm 9.

Algorithm 6 MTTKRP_K

```
1: procedure MTTKRP_K(($\mathbf{W}_C, \mathbf{A}, \mathbf{B}, \mathbf{X}_{(3)}, j, I, K$))
2:     Initialize $\mathbf{A}_{local}$, $\mathbf{X}_{local}$, $result = \mathbf{W}_C(global\_row, global\_col)$
3:     set $Tile = 32$, $b = $ floor$(I/Tile)$, $tx = threadIdx.x$, $ty = threadIdx.y$, $bx = blockIdx.x$, $by = blockIdx.y$, $global\_row = bx*TILE+tx$, $global\_col = by*TILE+ty$
4:     for t=0...b-1 do
5:         set $local\_row = t*TILE+tx$, $local\_col = t*TILE+ty$
6:         if $global\_row < K$ and $local\_col < I$ then
7:             $\mathbf{X}_{local}(ty, tx) = \mathbf{X}(global\_row, local\_col + j*K)$
8:         else
9:             $\mathbf{X}_{local}(ty, tx) = 0$
10:        end if
11:        if $local\_row < I$ and $global\_col < R$ then
12:            $\mathbf{A}_{local}(ty, tx) = \mathbf{A}(local\_row, global\_col)$
13:        else
14:            $\mathbf{A}_{local}(ty, tx) = 0$
15:        end if
16:        syncthreads()
17:        for $k = 0...Tile - 1$ do
18:            $result = result + \mathbf{X}_{local}(ty, tx) * \mathbf{A}_{local}(ty, tx)$
19:        end for
20:        syncthreads()
21:    end for
22:    if $global\_row < K$ and $global\_col < R$ then
23:        $\mathbf{W}_C(global\_row, global\_col) = result * \mathbf{B}(j, global\_col)$
24:    end if
25: end procedure
```

Algorithm 7 MTTKRP_I

```
 1: procedure MTTKRP_I(($\mathbf{W}_C, \mathbf{A}, \mathbf{B}, \mathbf{X}_{(3)}, j, I, K$))
 2:     Initialize $\mathbf{A}_{local}$, $\mathbf{X}_{local}$, $result = \mathbf{W}_C(global\_row, global\_col)$
 3:     set $Tile = 32$, $b = \text{floor}(I/Tile)$, $tx = threadIdx.x$, $ty = threadIdx.y$, $bx = blockIdx.x$, $by = blockIdx.y$, $global\_row = bx*TILE+tx$, $global\_col = by*TILE+ty$
 4:     for t=0...b-1 do
 5:         set $local\_row = t * TILE + tx$, $local\_col = t * TILE + ty$
 6:         if $global\_row < K$ and $local\_col < I$ then
 7:             $\mathbf{X}_{local}(ty, tx) = \mathbf{X}(global\_row, local\_col + j * K)$
 8:         else
 9:             $\mathbf{X}_{local}(ty, tx) = 0$
10:         end if
11:         if $local\_row < I$ and $global\_col < R$ then
12:             $\mathbf{A}_{local}(ty, tx) = \mathbf{A}(local\_row, global\_col) * \mathbf{B}(j, global\_col)$
13:         else
14:             $\mathbf{A}_{local}(ty, tx) = 0$
15:         end if
16:         syncthreads()
17:         for $k = 0 ... Tile - 1$ do
18:             $result = result + \mathbf{X}_{local}(ty, tx) * \mathbf{A}_{local}(ty, tx)$
19:         end for
20:         syncthreads()
21:     end for
22:     if $global\_row < K$ and $global\_col < R$ then
23:         $\mathbf{W}_C(global\_row, global\_col) = result$
24:     end if
25: end procedure
```

Algorithm 8 Compute $\mathbf{X}_{(1)}^{part}$

```
1: procedure ComputePartialMTTKRP($\mathbf{X}_{(1)}$,$C$,$count$)
2:     In parallel, for $gpu\_id = 0, ..., count - 1$, do
3:         for $j = gpu\_id\frac{J}{count} + 1, ..., (gpu\_id + 1)\frac{J}{count}$
4:             $[\mathbf{X}_{(1)}^{part}]_{((j-1)I:jI,:)} = \mathbf{X}_{(j)}^{1} C$
5:         end for
6:     end parallel for
7:     return $\mathbf{X}_{(1)}^{part}$
8: end procedure
```

Algorithm 9 Compute Parallel MTTKRP $\mathbf{W}_C$

```
1:  procedure ComputeParallelMTTKRP($\mathbf{X}_{(3)}$,$B$,$A$,$count$)
2:      set $\mathbf{W}_C = zeros(K, R)$
3:      In parallel, for $gpu\_id = 0, ..., count - 1$, do
4:          if $K > I$ then
5:              for $j = gpu\_id(J/count)...(gpu\_id + 1)(J/count) - 1$ do
6:                  MTTKRP_K($\mathbf{W}_C^{gpu\_id}, \mathbf{A}, \mathbf{B}, \mathbf{X}_{(3)}, j, I, K$)
7:              end for
8:          else
9:              for $j = gpu\_id(J/count)...(gpu\_id + 1)(J/count) - 1$ do
10:                 MTTKRP_I($\mathbf{W}_C^{gpu\_id}, \mathbf{A}, \mathbf{B}, \mathbf{X}_{(3)}, j, I, K$)
11:             end for
12:         end if
13:     end parallel for
14:     In parallel, for $gpu\_id = 0, ..., count - 1$, do
15:         $\mathbf{W}_C = \mathbf{W}_C + \mathbf{W}_C^{gpu\_id}$
16:     end parallel for
17:     return MTTKRP $\mathbf{W}_C$
18: end procedure
```

4.5 Speedups-Experiments

In this section, we present results obtained from the parallel CUDA implementation of ALS .

4.5.1 Setup

As we mentioned in Chapter 3, the experiments are carried out on a GPU accelerated node of the ARIS supercomputer. This node consists of a DELL PowerEdge R730 system with processor type Haswell - Intel(R) Xeon(R) E5-2660v3 (44 nodes in total - 2 sockets per node - 10 cores per socket), 64 GB RAM per node and 2 NVIDIA Tesla K40 GPUs per node.

4.5.2 Numerical Experiments

Tensors are generated from synthetic data, as described in previous chapter. For the sake of comparison, we also developed a version that solves the modified CPD–ALS and is executed only on CPUs. The following results are obtained after conducting 5 independent trials using the same input data.

In the following figures, we plot the execution time and the corresponding speedup obtained by using a serial CPU, a single–GPU and a dual–GPU implementation that solves the modified CPD-ALS. In Figures 4.5 - 4.8 we used a $1200 \times 1200 \times 1200$ cubic tensor of rank $R = 100, 200$ and in Figures 4.9 - 4.12 we used a $1500 \times 500 \times 500$ tensor of rank $R = 40, 200$.

We observe that the obtained speedup increases according to dimensions and rank R of the tensor. We attribute this fact to the computational cost, which as increases, it overcomes the respective transferring cost from and to device.

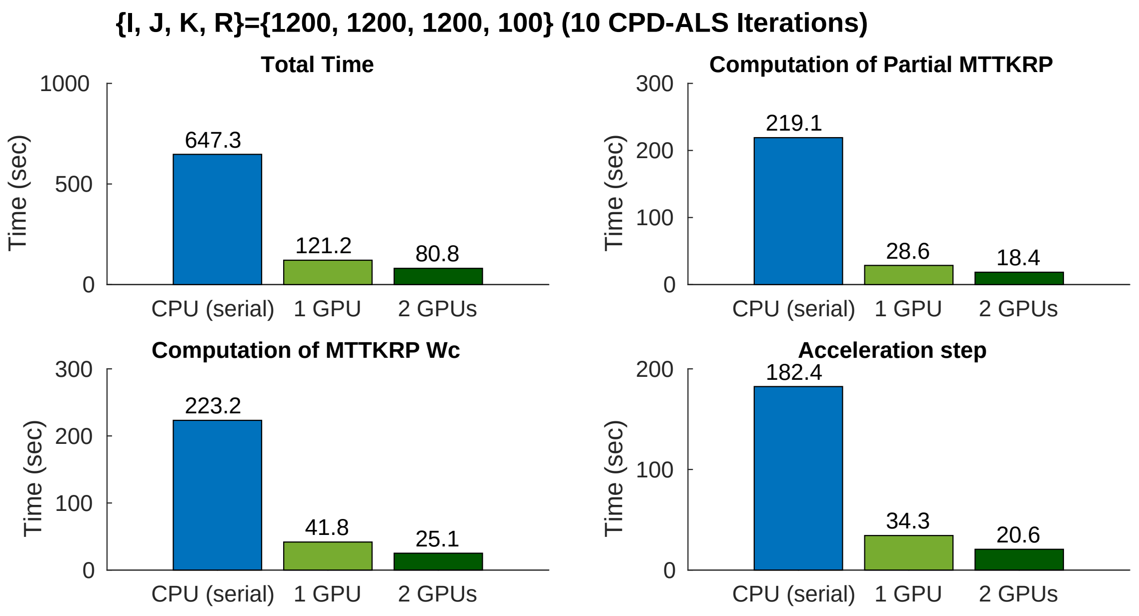

Figure 4.5: Experiment 1.A: 1200 × 1200 × 1200 tensor of rank $R = 100$

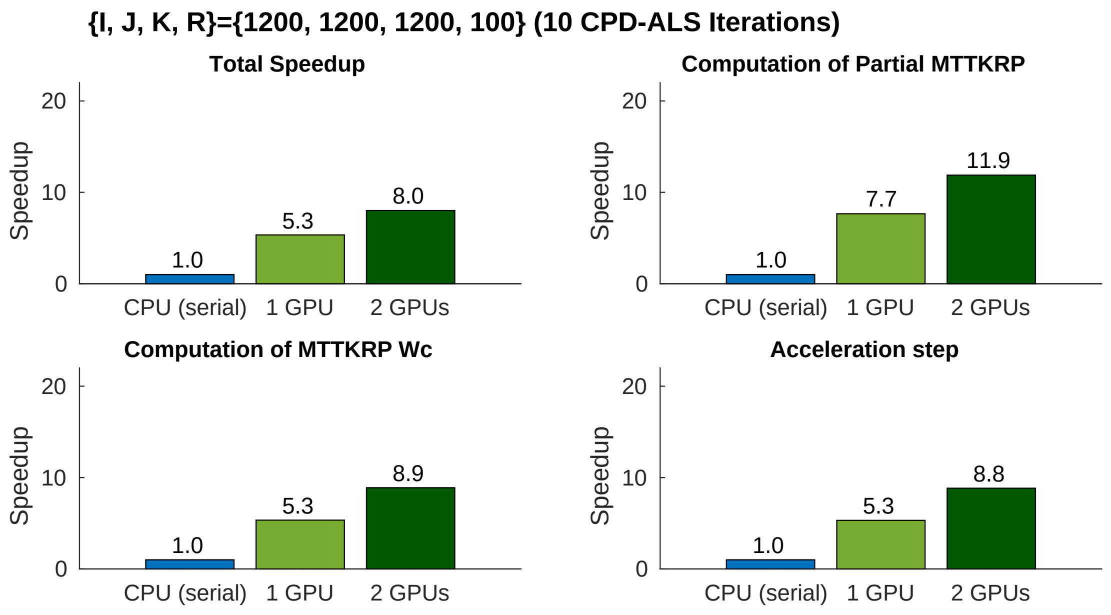

Figure 4.6: Experiment 1.A: 1200 × 1200 × 1200 tensor of rank $R = 100$

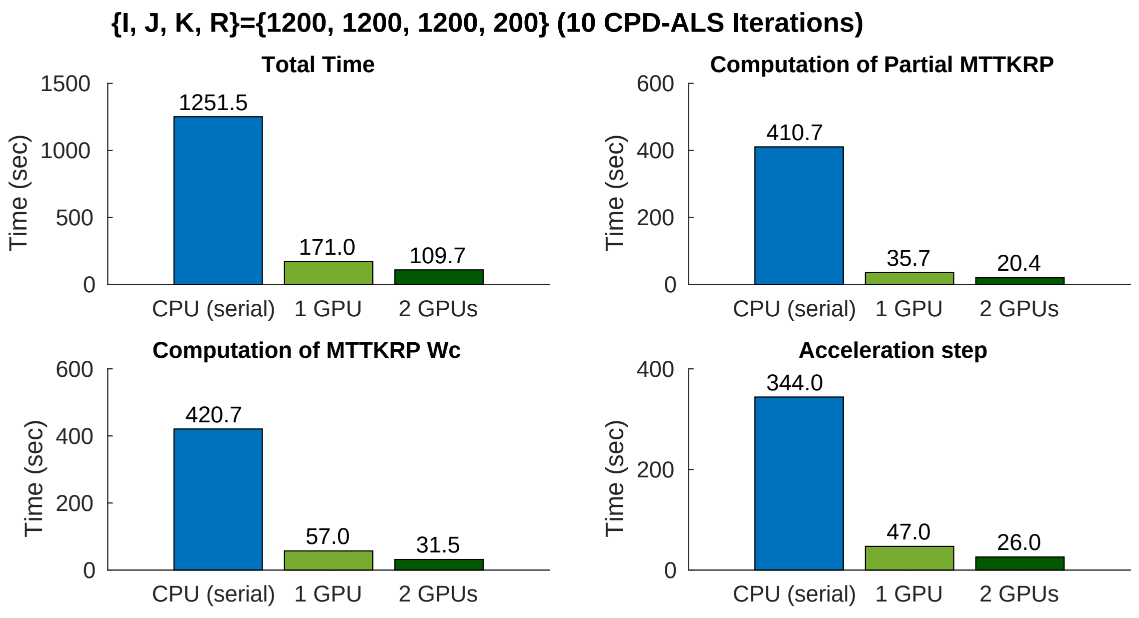

Figure 4.7: Experiment 1.B: $1200 \times 1200 \times 1200$ tensor of rank $R = 200$

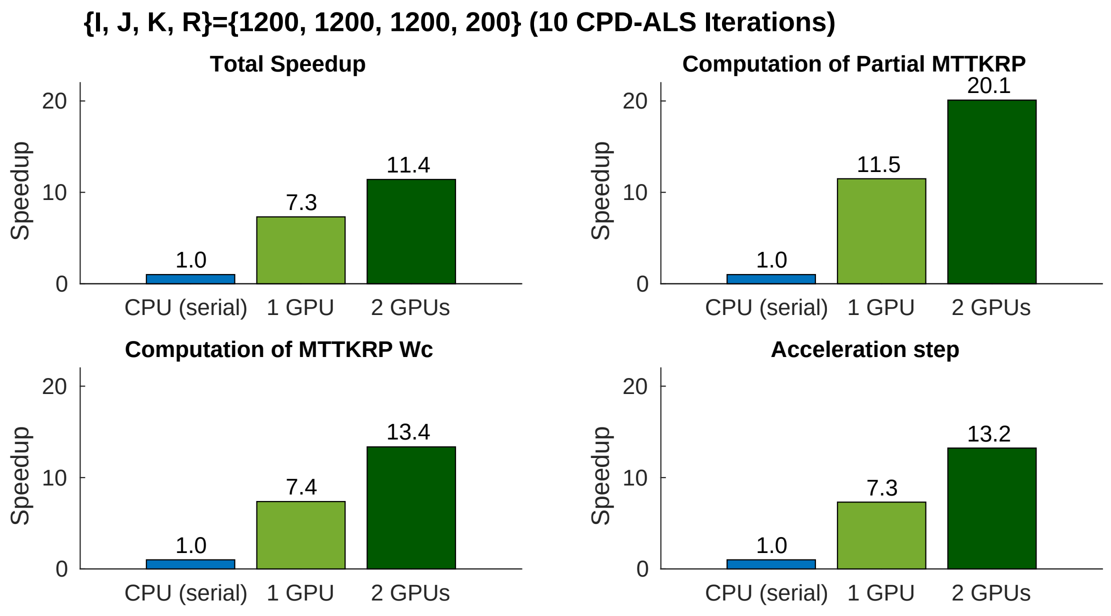

Figure 4.8: Experiment 1.B: $1200 \times 1200 \times 1200$ tensor of rank $R = 200$

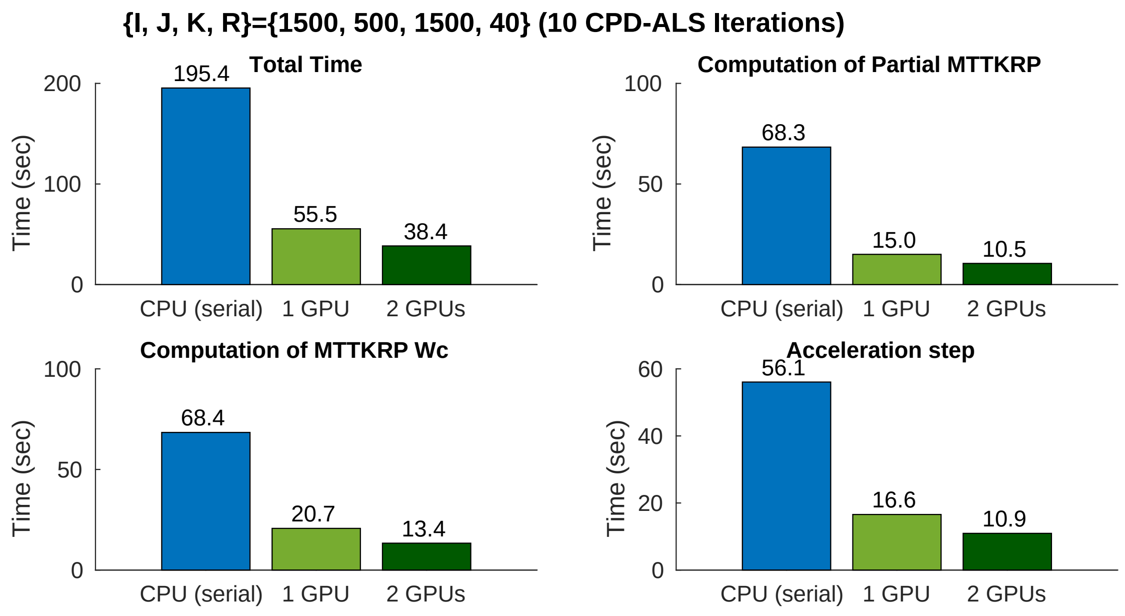

Figure 4.9: Experiment 2.A: $1500 \times 500 \times 1500$ tensor of rank $R = 40$

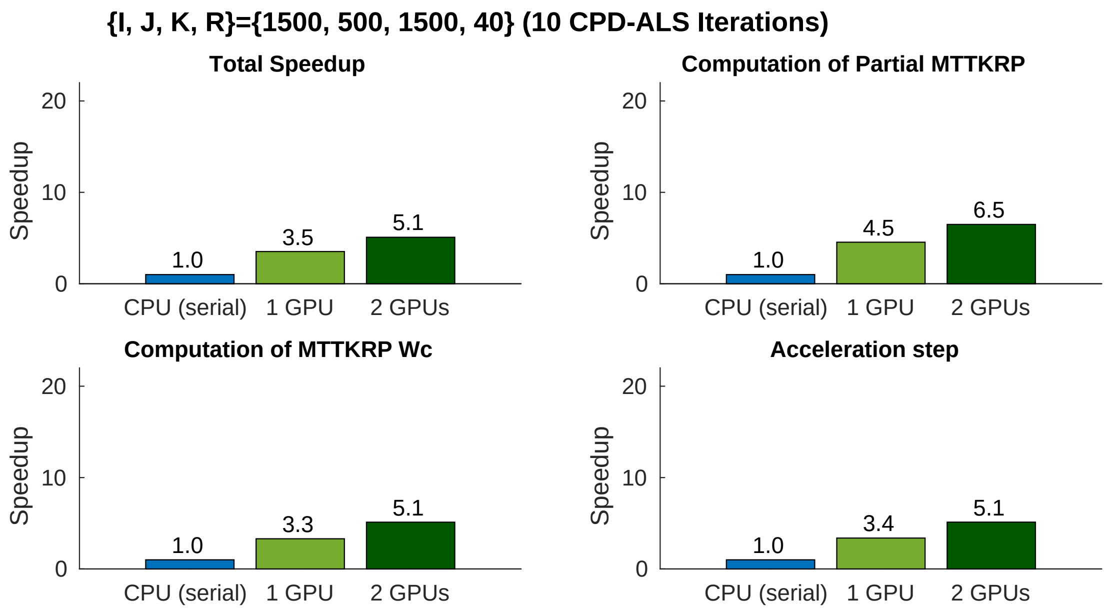

Figure 4.10: Experiment 2.A: $1500 \times 500 \times 1500$ tensor of rank $R = 40$

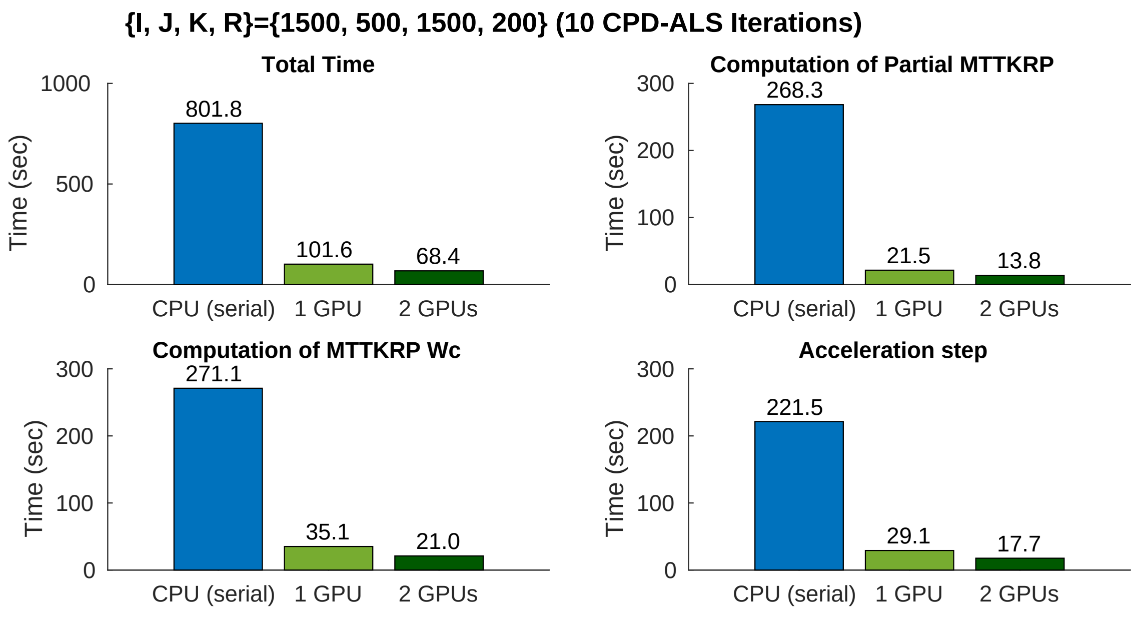

Figure 4.11: Experiment 2.B: $1500 \times 500 \times 1500$ tensor of rank $R = 200$

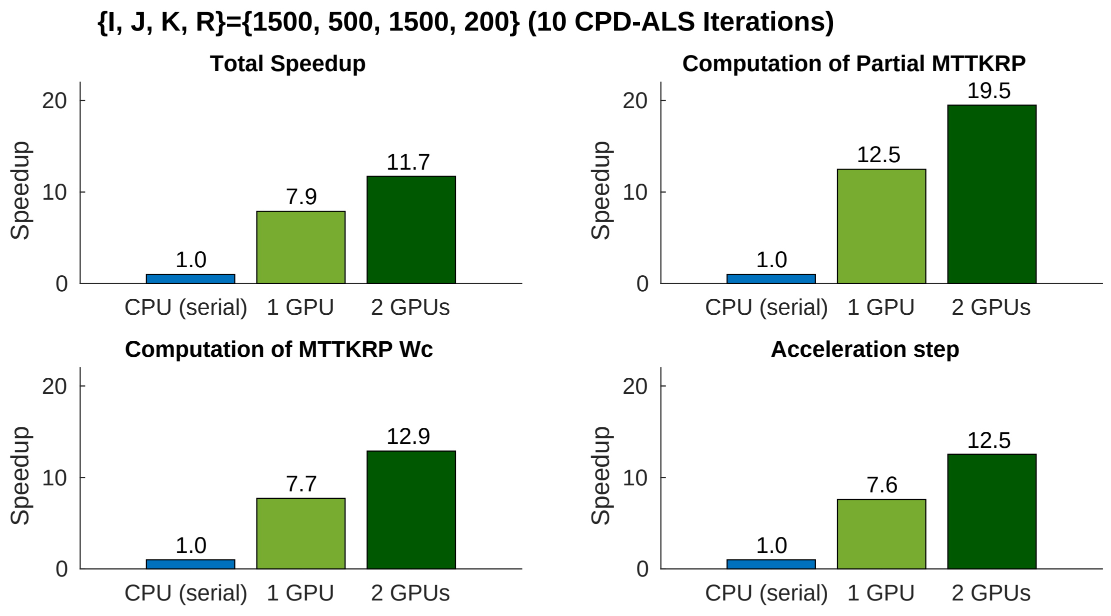

Figure 4.12: Experiment 2.B: $1500 \times 500 \times 1500$ tensor of rank $R = 200$

Chapter 5

Conclusion and Future Work

We considered the CPD model for tensor rank factorization via ALS and RBS–CPD methods. We used two APIs that support parallelism in modern heterogeneous & shared memory systems and described in detail parallel implementations of the aforementioned methods. In extensive numerical experiments, our parallel implementations were proven very efficient and attained significant speedup.

Since tensor factorization is a very useful and popular tool, we suggest some ideas for future work. One could impose constraints on factors, such as nonnegativity, sparsity, and orthogonality. For some applications, it might be also usable to extend to higher tensor orders and also try other tensor factorization models, such as Tucker (also known as *Multilinear SVD*), PARAFAC2, etc.

Finally, in order to handle and decompose large tensors faster, other stochastic frameworks could be also considered.

Bibliography

[1] M. Morup, “Applications of tensor (multiway array) factorizations and decompositions in data mining,” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 24–40, 2011.

[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, pp. 16:1–16:44, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2915921

[3] S. Rabanser, O. Shchur, and S. Gnnemann, “Introduction to tensor decompositions and their applications in machine learning,” 11 2017.

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

[5] A. P. Liavas, G. Kostoulas, G. Lourakis, K. Huang, and N. D. Sidiropoulos, “Nesterov-based alternating optimization for nonnegative tensor factorization: Algorithm and parallel implementation,” *IEEE Transactions on Signal Processing*, vol. 66, no. 4, pp. 944–953, Feb 2018.

[6] M. Rajih and P. Comon, “Enhanced line search: A novel method to accelerate parafac,” in *2005 13th European Signal Processing Conference*, Sep. 2005, pp. 1–4.

[7] N. Vervliet and L. De Lathauwer, “A randomized block sampling approach to canonical polyadic decomposition of large-scale tensors,” *IEEE Journal of Selected Topics in Signal Processing*, vol. 10, no. 2, pp. 284–295, March 2016.

[8] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” *Proc. of COMPSTAT*, 01 2010.

[9] P. Pacheco, *An Introduction to Parallel Programming*, 2011.

[10] OpenMP Architecture Review Board, “Openmp application program interface,” Specification, 2015. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[11] L. Huang, H. Jin, L. Yi, and B. Chapman, “Enabling locality-aware computations in openmp,” *Sci. Program.*, vol. 18, no. 3-4, pp. 169–181, Aug. 2010.

[12] T. M. John Cheng, Max Grossman, *Professional CUDA C Programming*, 2014.